1 2 9 0

UNIVERSIDADE D
COIMBRA

University of Salerno

Department of Computer Science

Master's Thesis in

Cybersecurity

A Zero-Knowledge-based system for vulnerability disclosure through the generation of Proofs of Exploits found in Solidity Smart Contracts

Supervisors

Prof. Christiancarmine Esposito

Prof. Naghmeh Ivaki

Prof. Nuno Laranjeiro

Candidate

Alessandro Cavaliere

Academic Year 2024/2025

Abstract

Vulnerability disclosure is a very complex process in the field of blockchain and cybersecurity. While software manufacturers want to protect their systems and preserve their reputations, vulnerability researchers want to prove that security problems exist. However, communication between researchers and companies can be difficult, as not all companies react promptly or offer adequate rewards for discoveries, creating frustration among researchers. This friction can lead to tensions in the disclosure process, with some companies choosing to ignore security reports or even fail to acknowledge potential risks. In the context of decentralized systems such as DeFi, these problems are even more severe. The open and immutable nature of smart contracts and the absence of centralized control further complicates the vulnerability management process. Bug-bounty programmes, for instance, are intended to be a cornerstone of responsible disclosure, providing researchers with a legal, compensated path to report flaws and giving project teams a structured workflow for triage and patching; yet many of these programmes are poorly managed, with some companies failing to honour rewards or offering token compensation, which discourages researchers from submitting critical vulnerabilities and leaves serious threats unaddressed. The situation is exacerbated by an influx of low-quality, AI-generated reports or bot spam that overwhelm triage teams and obscure legitimate findings.

A zero-knowledge approach can restore trust: by equipping researchers with a tool that produces a succinct proof showing that a specific transaction triggers an unintended state change, without revealing the exploit's technical details, both parties gain. Researchers can demonstrate exploitability and claim a bounty without risking premature disclosure, while project owners receive irrefutable evidence of risk and can release rewards promptly, confident that sensitive information remains hidden until a fix is deployed.

This thesis presents *zkpoex*, a ready-to-use Rust framework that generates zero-knowledge *proofs of exploit*. With a single command, the tool replays a confirmed exploit transaction, the exact calldata and value that cause the bug inside RISC Zero's zkVM, checks user-defined safety conditions, and produces a compact receipt that can be verified on-chain. In practice, a researcher can prove that their malicious transaction really breaks the contract and immediately claim the bounty, all without disclosing the underlying attack code; Meanwhile, the project gains cryptographic proof of the bug before any details leak, turning bounties into trustless, automatic agreements and raising the bar for collaborative vulnerability management in web3.

Contents

List of Figures

Code Listings

Introduction

1. Context and Motivation

Over the past decade, blockchain technology has evolved from a fringe experiment to a multi-billion-dollar industry. Yet security has not kept the same pace: in calendar year 2024 alone, *more than $2 billion in value were irreversibly lost because of smart-contract hacks and protocol exploits, with DeFi*[1] *as the primary target of crypto hacks.* In the picture 1, from a report of Chainalysis [10], you can observe the rising trend of stolen funds and hacks in decentralized ecosystems.

Attackers enjoy asymmetric advantages; they can remain anonymous, watch code evolve in public, and strike at the most opportune moment. At the same time, honest security researchers must reveal an exploit to prove its existence and then hope that a project honors its bounty commitment. Traditional bug-bounty workflows, therefore, introduce two fundamental frictions:

1. **Trust asymmetry**: the researcher must fully disclose the exploit before payment is guaranteed, risking front-running or litigation if the vulnerability is down-played.

2. **Operational overhead**: Projects drown in low-quality or spam reports and often need third-party escrow services to arbitrate payments.

Zero-Knowledge Proofs of Exploit (zkpoex) break this deadlock. By committing to a formal specification of *correct* state transitions and then proving, in zero-knowledge, that a private input violates at least one of those invariants, a white-hat hacker can demonstrate the presence of a flaw without disclosing the exploit itself. The open-source framework *zkpoex* [11] extends

[1] DeFi (Decentralized Finance) refers to financial services built on blockchain networks, eliminating intermediaries through smart contracts.

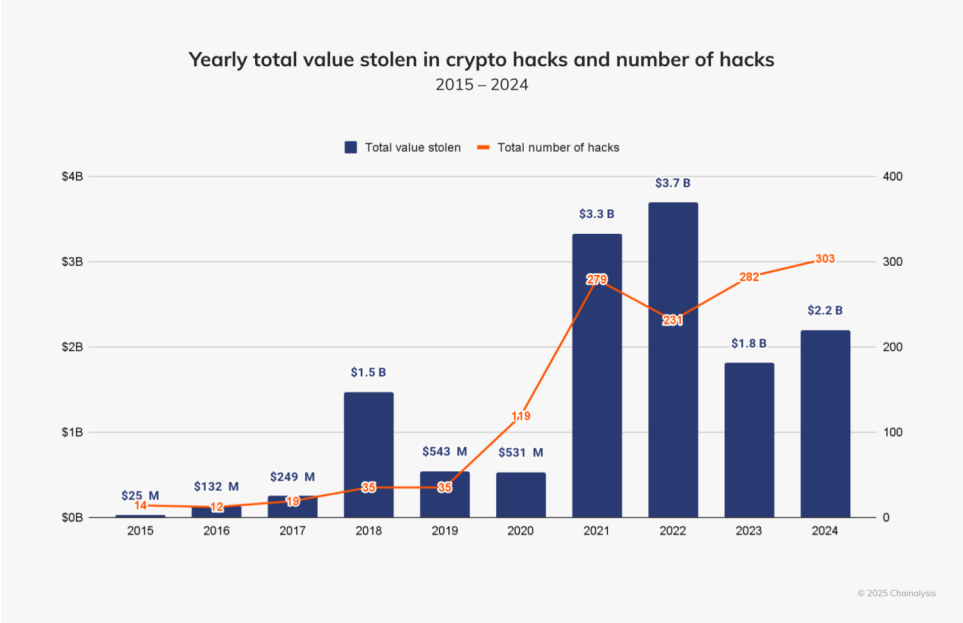

Figure 1: Yearly total value stolen in crypto hacks and number of hacks from 2015 to 2024.

this paradigm, combining a RISC-V-based[2] zkVM[3] with an Ethereum interpreter so that any Solidity contract can be audited in a wholly trustless setting. The vision is compelling: automatic bounty pay-outs, instant verifiability, and no leak of exploit details, exactly the properties that classical disclosure channels lack.

2. Problem Statement

Blockchain security today suffers from a paradox: vulnerabilities are straightforward to exploit yet notoriously hard to disclose safely. Smart-contract authors need cryptographic assurance that a reported flaw is genuine before acting, while ethical hackers need economic assurance that disclosure will be rewarded rather than front-run. Bridging this chasm demands a protocol able to transform the *existence* of an exploit into a *verifiable* mathematical

[2]RISC-V is an open-source, modular, and royalty-free instruction set architecture (ISA) designed to be simple, extensible, and suitable for everything from microcontrollers to supercomputers.

[3]zkVM (Zero-Knowledge Virtual Machine) is a virtual machine that executes computations while generating zero-knowledge proofs of their correctness. This enables trustless verification of program execution, a key concept in blockchain scalability and privacy. A detailed discussion follows in Section 1.6.

statement, one that can be checked on-chain without revealing the exploit itself. The remainder of this section formalises that challenge and breaks down the system requirements that drive *zkpoex* design.

From "exploitable" to "provably exploitable"

Smart-contract audits ultimately ask a binary question:

Does there exist a calldata c that induces an invalid state transition in the contract's execution?

Finding such an input is only half the battle; *convincing the contract owner without revealing it* is the harder half. This dilemma crystallizes into three interlocking technical challenges:

C1: **Specification Gap**: Real-world contracts rarely come with a executable *specification*. Without formal post-conditions (e.g. invariants on balances, total supply, access-control flags), a proof system has nothing to assert against.

C2: **Zero-Knowledge Execution**: Even with a clear specification in hand, we still have to replay the contract call inside a zero-knowledge proving system without exposing any data. Building a huge custom circuit that hard-codes every single EVM instruction would be far too slow and complex. A better approach is to use a general-purpose zero-knowledge virtual machine (zkVM) and run an ordinary EVM interpreter inside it; this keeps the proof trace compact and manageable for everyday transactions.

C3: **Trustless Settlement**: Once a proof is generated, payment must be *automatic*. Any off-chain escrow would reintroduce trust and timing risk, so the verification logic must live on-chain, remain gas-efficient, and emit side effects (bounty payout, pausing the contract, and related actions) atomically.

Why existing approaches fall short

Prior art demonstrates zero-knowledge proofs of exploitability for firmware [12] and LLVM[4]binaries [13] (the papers are detailed in Section 4. of the Introduction Chapter); however, none address the peculiarity of smart contracts. To the best of our knowledge, this thesis provides the first scientific

[4]LLVM (Low Level Virtual Machine) is a compiler infrastructure that produces binaries and intermediate code representations used for optimization and cross-platform compilation across various programming languages.

and academic documentation of zero-knowledge proofs of exploitability in the context of smart contract security. Below, we outline two fundamental challenges that distinguish the smart contract setting from previous works:

- **State Explosion**: Unlike ordinary programs, a smart contract's execution depends on persistent, on-chain state (e.g., user balances, contract storage). This state is stored in a Merkle-Patricia trie, a cryptographic data structure that allows efficient verification of key-value pairs. To prove an exploit, the prover must reproduce just the slice of the global state that the transaction touches, otherwise, the proof grows enormous. Hence the term *state explosion* [14].
- **Trustless Finality**: On-chain attacks finish in a single block, so the fix (paying the bounty and optionally pausing the contract) must also complete in one atomic transaction. Any scheme that needs off-chain arbitration reintroduces the very trust issues we are trying to remove.

Formalisation of the Problem

Formally, let:

$$\mathcal{M} : \big(s,\, c\big) \;\mapsto\; s'$$

be the EVM transition function that maps an initial state s and calldata c to a new state s'. Let $\Phi(s, s')$ denote a decidable predicate capturing all contractual invariants. The *zkpoex* problem is to design a protocol where a prover convinces a verifier of the statement

$$\exists\, c \;.\; \neg\Phi\big(s, \mathcal{M}(s, c)\big)$$

without revealing c and such that the following properties hold:

(a) **Soundness**: No adversary can convince an honest verifier unless a real exploit exists.

(b) **Zero-Knowledge**: The verifier learns nothing about c beyond the fact that an exploit exists.

(c) **Atomicity**: Upon acceptance, the bounty is disbursed within the same Ethereum transaction that verifies the proof.

The formula says:

> "There exists a calldata c such that executing c from the current blockchain state s produces a successor state s' where the **state transition** $(s \to s')$ violates at least one contract rule Φ."

Solving the problem above removes the fundamental trust bottleneck that plagues traditional vulnerability disclosure and lays the groundwork for *specification-driven* smart-contract development, where proving a bug becomes as routine as running unit tests.

3. Research Objectives

Although individual research papers have demonstrated proofs-of-concept(PoC) for *software* binaries, the literature lacks a system that is *tailor-made for blockchain* and that leverages modern, transparent proof systems. Consequently, this thesis pursues four concrete objectives:

1. **Design** a protocol that allows a prover to convince a verifier that a smart-contract exploit exists *without* revealing calldata, bytecode patches, or private keys.
2. **Implement** an end-to-end prototype: `zkpoex v0.1.0`, built on `Risc0` [15] and `Rust-EVM` [16], capable of handling realistic scenarios.
3. **Evaluate** the performance, scalability and security assumptions of `zkpoex v0.1.0` through extensive benchmarking and profiling (e.g. proving time, proof size or cycle counts), *without* direct comparison to external toolchains.
4. **Validate** the protocol with real-world case studies on public Ethereum testnets, reporting empirical results proving time, on-chain verification gas, and bounty-payout latency, to substantiate the effectiveness of the proposed approach, again *without* cross-tool-chain benchmarks.

4. Research Methodology

A `Systematic Literature Review (SLR)` was carried out to chart current research on Zero-Knowledge Proofs of Exploitability (or Exploit). The review was organised following Kitchenham's three-stage process [17]:

`(i)` **Planning**: Research questions were refined and trial search strings evaluated. An illustrative query was:

```
("vulnerability disclosure" OR "vulnerabilities disclosure")
AND blockchain AND ("ZKPs" OR "Zero Knowledge Proofs"
OR "ZKP") -"IoT" -"machine learning"
```

`(ii)` **Conducting**: The engines *IEEE Xplore*, *ACM Digital Library*, *arXiv*, and *Google Scholar* were searched for publications dated 2013–2025.

(iii) **Reporting**: Automatic filters were first applied, then a manual screening followed. Titles and abstracts were checked against *inclusion* criteria (blockchain relevance, demonstrable exploit, peer-review, English language) and *exclusion* criteria (IoT or ML focus, topics unrelated to vulnerability disclosure, grey literature lacking technical substance). Surviving papers underwent full-text review; quality was logged on a five-point Likert scale covering rigour, reproducibility and relevance. The process yielded **two** peer-reviewed primary studies; an additional survey paper and an open-source prototype were retained as complementary evidence that sharpens the problem framing but do not themselves constitute peer-reviewed solutions.

Primary studies

1. **Hardware-oriented proof generation: *Efficient Proofs of Software Exploitability for Real-world Processors*** [12]

   The study targets the MSP430, a 16-bit microcontroller family often embedded in low-power devices such as smart meters and sensor nodes. The authors model the entire instruction set in *Verilog*, a hardware-description language that can be compiled into an arithmetic circuit suitable for a zk-SNARK back-end. zk-SNARKs (Zero-Knowledge Succinct Non-Interactive Arguments of Knowledge, which will be introduced together with the basis of ZKPs and related history in Section 1.5), briefly, allow a prover to convince a verifier that a computation was executed correctly without revealing intermediate data, while keeping both proof size and verification time small. By reducing gate count and re-ordering constraints, the paper shows that one can generate a proof of exploitability whose cost is linear in the execution trace, rather than logarithmic. In practical terms, this means that an attacker could demonstrate control-flow hijacking on a firmware image without disclosing the payload, and the vendor could check the proof in milliseconds. The work, therefore, supplies a concrete example of how verifiable proofs-of-exploit can enable fair disclosure in resource-constrained environments.

2. **Compiler-driven circuit synthesis: *Cheesecloth: Zero Knowledge Proofs of Real-World Vulnerabilities*** [13]

   Cheesecloth starts from LLVM Intermediate Representation (LLVM IR), the language-agnostic bytecode to which modern compilers lower C, C++, Rust, and other high-level languages. It then maps LLVM IR into *MicroRAM* constraints, a simplified random-access machine

model widely adopted by zk toolchains, so that memory accesses become explicit Boolean conditions. The framework encodes classic memory-safety properties such as buffer bounds and use-after-free checks, and it proves violations inside libswanky's Mac-n-Cheese [18] prover without exposing the exploit inputs. From a broader standpoint, the paper illustrates how compiler infrastructure can bridge the gap between rich source languages and the restricted algebraic models accepted by zero-knowledge proof systems. The strategy of compiling a real execution trace into a minimalist RAM abstraction is directly relevant to smart-contract settings, where EVM bytecode can be lifted to a RISC-V or MicroRAM dialect before proof generation.

Complementary evidence

- **Applicability survey: *Do You Need Zero-Knowledge Proofs?* [19]**
  This IACR ePrint (2024/050) offers a decision framework for privacy technologies. Section 4.3, "Proving Software Exploits", highlights the same disclosure dilemma tackled here and cites the *ZkPoEX* [20] repository as the only public attempt at a zero-knowledge exploit proof in the blockchain ecosystem.

- **Open-source prototype: *ZkPoEX* (ETH Denver 2023) [20]**
  Though not peer-reviewed, *ZkPoEX* wraps an unmodified EVM interpreter inside RISC Zero and emits a recursive STARK attesting to an invariant failure while hiding calldata and keys. Its architecture inspired the prototype in this thesis; however, obsolete dependencies and an outdated Rust toolchain required a complete re-implementation. Two lessons stand out:

  (i) General-purpose *zkVMs* can execute contract bytecode unaltered, obviating bespoke circuits.

  (ii) On-chain verification removes the need for an external escrow in bounty negotiations.

- **Bug Bounty Trilemma: Beyond the Bug Bounty Programs Trilemma: Bounty 3.0's Blockchain-ZKP Approach [21]**
  Although it proposes no exploit-proof system, the paper articulates the trust asymmetry between finder and vendor. It envisages a token-incentivised, zk-enabled escrow where blockchain immutability records disclosure events and zero-knowledge proofs safeguard payload secrecy. The design rationale directly mirrors the problem solved by our prototype.

Collectively, the two peer-reviewed studies supply technical foundations, while the survey, the prototype, and the Bug Bounty 3.0 paper sharpen the application context, confirming both the relevance of our research questions and the novelty of pursuing zero-knowledge exploit proofs for smart contracts.

5. Document Structure

The remainder of the dissertation is organised as follows:

- **Chapter 1: Background and State of the Art**
  Surveys the architecture of blockchain systems, the structure of Ethereum transactions and smart contracts, and the theoretical underpinnings of zero-knowledge proofs and zkVMs.
- **Chapter 2: Proposed System Architecture**
  Describes the end-to-end design of the `zkpoex` framework: integration of the RISC0 zkVM, interoperability layers with the `Rust EVM`, and the exploit execution flow leading to proof generation.
- **Chapter 3: Implementation and Validation**
  Details the development environment (Rust + Risc0, and related tooling), the *VerifierContract* deployed on Ethereum Holesky, and analytical case studies covering common smart-contract vulnerabilities such as reentrancy attack.
- **Chapter 4: Performance Evaluation and Results**
  Benchmarks `zkpoex` on representative exploits, summarising guest time, cycle counts, on-chain gas cost, and flame graph metrics for each case study.
- **Chapter 5: Conclusions and Future Perspectives**
  Reflects on the outcomes of the research, highlighting the main contributions of the `zkpoex` framework to smart-contract security; discusses the system's current limitations, evaluates its applicability in real-world disclosure scenarios, and presents potential directions for future work.

This logical progression: *problem* $\rightarrow$ *theory* $\rightarrow$ *architecture* $\rightarrow$ *empirical validation*, ensures that each chapter builds on the foundations laid by the previous one, guiding the reader from high-level motivation to concrete, experimentally verified results.

Chapter 1

Background and State of the Art

This chapter offers a compact map of the knowledge needed to follow the rest of the thesis. We start with the building blocks of public blockchains: hash functions, Merkle trees, digital signatures, and the two dominant consensus schemes, Proof-of-Work and Proof-of-Stake. Then, zoom in on Bitcoin's UTXO ledger and Ethereum's smart-contract account model, the specific terrain on which our exploits run. Next, we sketch the evolution of zero-knowledge proofs, from early interactive protocols to today's fast SNARK/STARK systems, before introducing zero-knowledge virtual machines (zkVMs), the technology that lets us prove an entire program's execution without revealing its data. These three threads provide the conceptual toolkit that the subsequent chapters will apply when presenting and evaluating *zkpoex*.

1.1 Brief Introduction to Blockchain

A blockchain can be understood as a publicly distributed digital ledger that maintains transactions among members of a network. These transactions are assembled into sequential blocks, each referencing the previous one according to specific rules and data structures. A pivotal aspect of blockchain technology is that after a block is appended to the chain, any subsequent alteration demands the modification of all blocks that follow it, making tampering highly impractical.

The conceptual origins of this technology can be traced to David Chaum's 1982 doctoral research at the University of California, titled *"Computer Systems Established, Maintained and Trusted by Mutually Suspicious Groups"* [22]. Subsequent exploration into a cryptographically protected chain of

blocks was documented in 1991 by Stuart Haber and W. Scott Stornetta [23], who aimed to establish a tamper-resistant mechanism for document timestamps. The following year, in collaboration with Dave Bayer, they enhanced the design by integrating Merkle trees, an improvement that allowed multiple document certificates to be batched into a single block, thereby increasing efficiency [24].

Later, Wei Dai's b-money proposal [25] introduced, in 1998, the idea of creating currency by solving computational challenges and contemplated achieving consensus through interactions among participants, though its implementation details remained abstract. In 2004, Hal Finney presented a functioning Proof-of-Work through the idea of "reusable proof of work" model [26], representing a gradual step forward. However, it was only in 2008 that a decisive leap occurred under the pseudonym Satoshi Nakamoto [27], who introduced Bitcoin. As a digital currency rooted in blockchain, Bitcoin offered the first thorough application of decentralized consensus and computational proofs. Cryptocurrencies remain one of the most significant implementations of blockchain, requiring close analysis of their underlying mechanisms and real-world potential.

Bitcoin was the first cryptocurrency to gain broad recognition, aiming to remove the need for a central intermediary in peer-to-peer transactions. In the wake of Bitcoin's advent, various other cryptocurrencies emerged, such as Ethereum, the blockchain on which this document focused for the development of *zkpoex*.

1.2 Foundational Concepts

Before delving into the technical details that follow, the reader is assumed to possess a working knowledge of the following foundations: P2P networks, structure of a block, cryptographic hash functions, Merkle trees, asymmetric cryptography with digital signatures, and consensus algorithms such as Proof-of-Work (PoW) and Proof-of-Stake (PoS). The subsections below develop these notions systematically, thereby offering a self-contained reference for the general concepts invoked throughout the thesis. Please note that the exposition that follows adopts *Bitcoin* as the canonical reference, especially for block layout and related data structures, because it remains the earliest, most thoroughly analysed, and pedagogically transparent public blockchain.

1.2.1 Peer-to-Peer Networks

A peer-to-peer (P2P) network is a particular example of a decentralized system. This concept is the paradigm followed by blockchains like Bitcoin and Ethereum. P2P networks consist of a group of computers or individuals

interacting with each other, having similar permissions and responsibilities when processing data. No device in this network exists whose sole function is to send or receive data. All of the network's *nodes*, which are the names given to its participants, are both clients and servers at the same time. To guarantee non-repudiation, privacy, and authentication, a decentralised system, such as a P2P network, must rely on a protocol like that detailed in Public-key cryptography section.

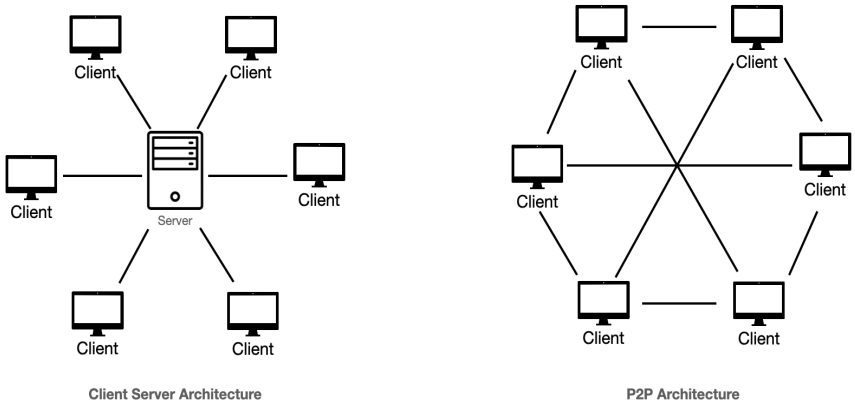

Figure 1.1: Client-Server Architecture and P2P Architecture compared.

1.2.2 Structure of a Block

To begin, we present a concise overview of the structure of an individual blockchain block. A block is divided into two distinct parts: the *header* and the *body*. The block *header*, illustrated in Figure 1.2, consists of six fields that govern the block itself:

- **Version**: specifies the *block-validation rules*[5]. The ruleset depends on the software version that drives the specific blockchain.
- **Parent Block Hash** (or **Previous Hash**): a 256-bit value that references the preceding block in the chain. If the previous block's hash changes, so too does the *Previous Hash*, triggering a cascade of updates across every subsequent block. See Section 1.2.3.
- **Merkle Tree Root Hash**: the root of a Merkle tree used, among other purposes, to guarantee data integrity via hashing (Section 1.2.6).

[5]Block validation is the essence of a blockchain. It allows the block to be appended to the chain only after a sequence of checks.

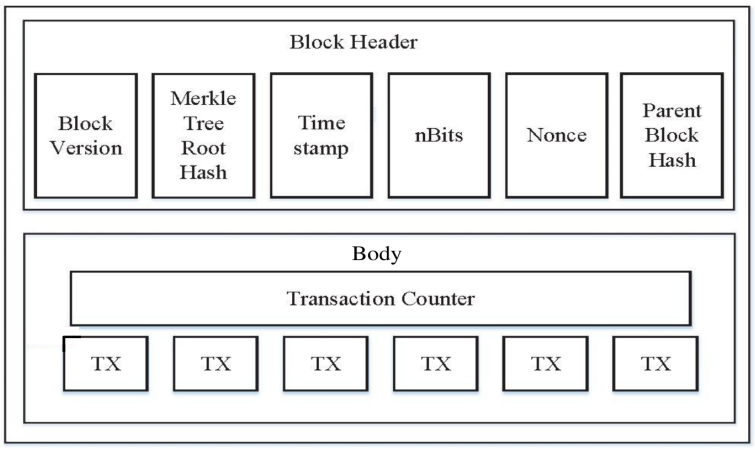

Figure 1.2: Structure of a blockchain block.

- **Timestamp**: the precise instant of block creation, certifying that the recorded events occurred at that moment.
- **Bits** (or **difficulty target**): the Proof-of-Work threshold that a valid block hash must not exceed.
- **Nonce**: a byte-level value repeatedly re-hashed[6] until the resulting hash meets the difficulty criterion. This computationally expensive search constitutes the well-known process of *mining*.

The block *body* consists of a *transaction counter* and the transactions themselves (denoted "TX" in Figure 1.2). Each transaction must be *verified*[7]. The verification logic is examined in Section 1.2.5.

1.2.3 Hashing for Block Linking

As noted above, a blockchain is a data structure composed of sequentially linked blocks (a minimal schematic appears in Figure 1.3). New blocks form whenever network participants create fresh data or update existing records. Each block stores cryptographically protected data together with a unique identifier: the block hash.

[6] Re-hashing denotes successive hashing operations on the evolving header.

[7] Verification confirms the details of the transaction, including timestamp, amount, and participants.

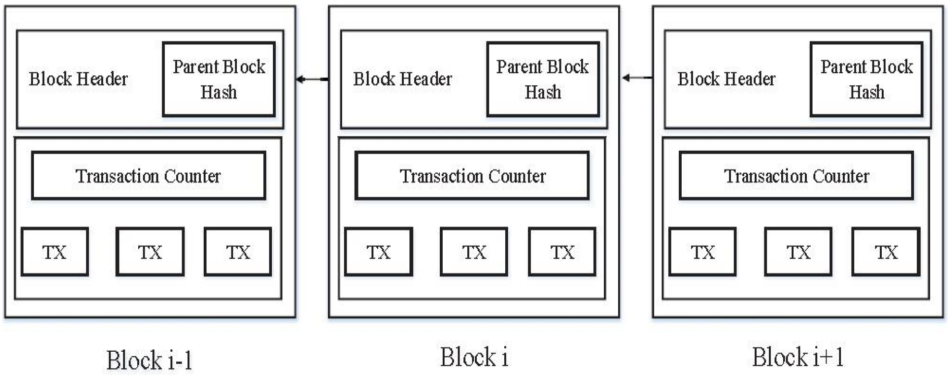

Figure 1.3: Minimal schematic of a blockchain.

The hash is the adhesive that binds blocks together, producing a genuine *chain of blocks*. By design, it renders the chain tamper-evident, ensuring both the integrity and authenticity of every block's data. This subsection introduces hashing in general and its specific application in blockchain technology.

Hashing is the process whereby an arbitrary key k is fed to a *hash function* h to obtain an integer $h(k)$. The key-value pair `<k,v>` is stored in a vector (the *hash table*) at position $h(k)$. Formally,

$$h : U \to \{0, 1, \ldots, m-1\}, \quad \text{where}$$

- h is the hash function;
- U is the Universe set of possible keys;
- $m \in \mathbb{N}$ is the size of the vector T;
- T is the hash table that stores the key-value pairs, exemplified in Figure 1.4 as *storage*.

Within a blockchain, *cryptographic hash functions*, a specialized subset of classical hash functions, must satisfy the following properties:

- *Determinism*: identical input always yields the same hash.
- *Collision resistance*: finding two distinct inputs that produce the same hash is computationally infeasible.
- *Pre-image resistance*: deriving the original message from its hash is computationally infeasible.
- *Efficiency*: the hash must be computable quickly.

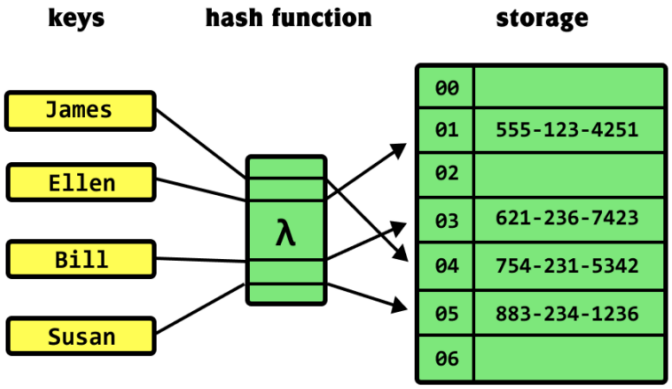

Figure 1.4: Example of a hash table.

These attributes make cryptographic hash functions ideally suited to guarantee integrity, authenticity, and data security.

Applied to blockchain, the hash endows each block with uniqueness. Because every block includes the hash of its immediate predecessor, any alteration to a block propagates forward, invalidating all successors. As highlighted in Section 1.2.2, each block also contains a *timestamp* that contributes to its hash and, in the words of [27], "[...] proves the data must have existed at that time, since it features in the hash."

There are numerous cryptographic hash functions, but over time, it has been shown that some of them have vulnerabilities. Today, the most widely employed family is the *Secure Hash Algorithm* (SHA). Collisions discovered in SHA-0 and SHA-1 render them ill-advised for new deployments, whereas more recent variants remain robust. Bitcoin employs SHA-256 (a SHA-2 family member) with a 256-bit digest. Ethereum, before *The Merge* upgrade of 15 September, used Keccak-256 with a variable-length digest; Keccak is the basis for SHA-3, the latest member of the SHA standard family.

1.2.4 Public-key cryptography

In 1976, Whitfield Diffie and Martin Hellman, working in concert with Ralph Merkle, introduced the concept of public-key (asymmetric) cryptography [28]. This revolutionary idea rests on mathematically linked pairs of keys (public and private key).

A *private key* is a randomly generated secret value known only to its creator. It is used:

(i) To decrypt any message that has been encrypted with the matching public key;

(ii) To generate a digital signature.

Every *private key* is associated with a unique *public key*, a number that can be distributed without restriction. Anyone may employ the public key to encrypt information intended for the key-pair owner or to verify signatures produced with the private key.

Cryptocurrencies dispense with a central authority to oversee transactions, but this raises the challenge of establishing trust among users. Consensus protocols resolve that difficulty by compelling the network's nodes to agree on which transactions are valid and by ensuring that once recorded, they remain immutable. *PoW* and *PoS* consensus protocols are mentioned respectively in 1.2.7 and 1.2.8.

1.2.5 Transactions: The Digital Signatures Chain Mechanism

In a blockchain, transactions exchange assets of any nature among two or more parties and must be broadcast to and subsequently stored by every network participant. Because the system is decentralised, any transaction enjoys the same significance as any other; hence, all nodes must verify its authenticity.

Satoshi Nakamoto proposes in [27] a system of *digital signatures*. Every peer-to-peer (P2P) node holds a *public key* and a *private key*:

- The *public key* (often denoted as an address) is visible to every network participant.
- The *private key* is known only to its owner.

To implement the digital-signature system, each emitted transaction is signed with the hash value (generated as described in Section 1.2.3) and the sender's private key. The resulting output is broadcast across the network using the recipient's public key. Only the recipient can decrypt the message, as only they possess the matching private key, and thus can verify the digital signature.

Once authenticity has been established, we must consider how transactions are stored within blocks. They are chained together by the *Digital Signatures Chain*; as [27] puts it, "[...]each owner, when transferring the coin, signs a hash of the previous transaction and the public key of the next owner[...]". The outcome is a genuine chain of transactions, stored inside every block; a graphical illustration appears in Figure 1.5. As indicated in Section 1.2.2, multiple transactions reside in each block body. Each belongs

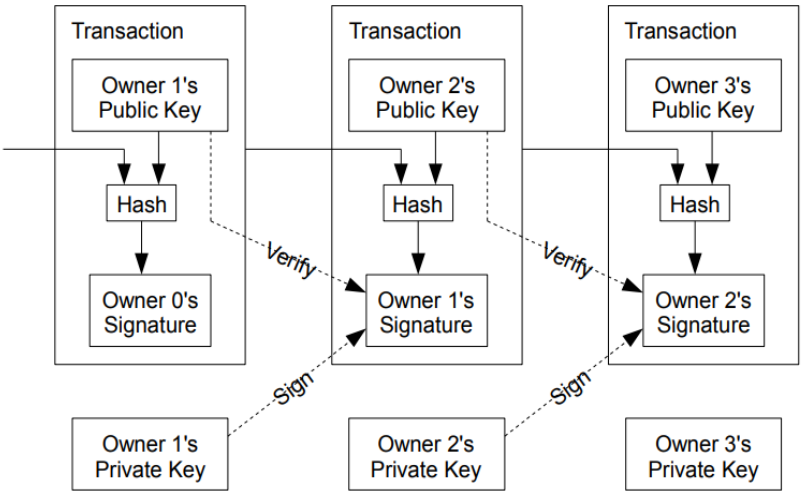

Figure 1.5: Transactions stored within a block via the Digital Signatures Chain mechanism.

to one or more transaction chains that collectively track the complete spending history of every participant's digital funds. In Bitcoin's white paper [27], the aggregated structure is termed *electronic cash*.

1.2.6 Merkle Tree Roots and Data Integrity

A further component that guarantees data integrity within a block is the *Merkle tree*, an *arborescent* data structure[8] whose single root (the Merkle Tree Root) is stored in the block header (see Section 1.2.2).

The bottom layer of this tree "[...] displays the transactions stored (e.g. T001 in Figure 1.6), which are subsequently converted into their SHA-256 hash digests (e.g. H001), forming the leaves of the Merkle tree [...]" [29]. The root is computed by recursively hashing upward from the leaves. If even a single node is modified, the resulting hash variation propagates to the root, making any tampering immediately detectable. Consequently, nodes use the Merkle root to verify that received blocks are intact and that no past transaction has been altered.

1.2.7 Proof-of-Work Mining for Block Validation

Having understood transaction authenticity, we now examine how one or more transactions are *validated*. Transactions must be broadcast to other

[8] A tree is a data structure composed of nodes linked by edges, yielding hierarchical organisation.

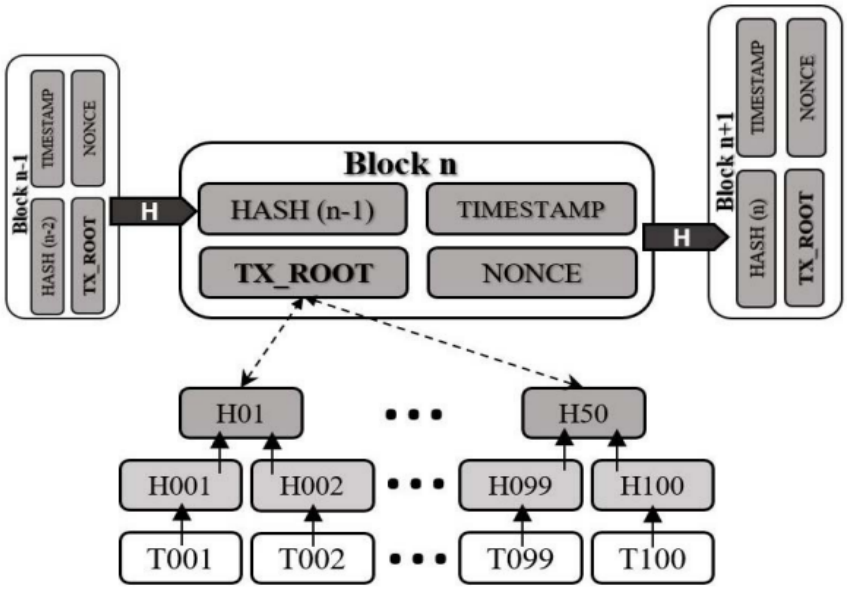

Figure 1.6: Merkle Tree Root stored in the block header.

network nodes for verification. In Bitcoin's mining model, specialised *validator nodes* (*miners*[9]) receive the pool of pending, unvalidated transactions and propose the new block to be published.

To validate transactions and assemble them into a new block, miners must compute a specific value, the *Nonce*, so that the resulting block hash satisfies a predetermined property. Recall that any change in the block's fields alters its hash. The computationally intensive task is precisely to adjust the Nonce until the hash meets the difficulty target.

Each block, together with its transaction history, must achieve a valid hash to be accepted. In Bitcoin, for instance, the criterion is that the block hash begins with a specific number of leading zeros. Only the first miner to solve the puzzle earns the block reward, a quantity of newly minted bitcoin generated by the process. Consequently, miners with greater computational resources enjoy a higher probability of capturing the reward.

Once mined, the block is appended to the blockchain, increasing its length, and all contained transactions are finalised.

[9]Miners are owners of machines that contribute computational power and energy to a Proof-of-Work cryptocurrency network.

1.2.8 Proof-of-Stake Consensus for Block Validation

In a Proof-of-Stake (PoS) system, the authority to extend the ledger derives from economic stake rather than raw computation, like in 1.2.7. Prospective *validators* lock coins in a deposit contract, exposing collateral that can be confiscated. As mentioned in [30]: "[...] One validator is randomly selected to be a block proposer in every slot. This validator is responsible for creating a new block and sending it out to other nodes on the network. [...]". The proposer assembles the pending transactions into a candidate block and broadcasts it together with a signature under her validator key; committee members independently verify the block and emit weighted attestations that are aggregated into its header.

Safety is underpinned by *slashing*: any validator that equivocates or signs an invalid block forfeits a portion of its staked capital, turning misbehaviour into a direct financial loss. As long as two/thirds of the staked value adheres to the protocol, liveness and finality are preserved. The algorithm used in proof-of-stake Ethereum is called *LMD-GHOST* [31], which overlays a longest chain rule with a Byzantine fault-tolerant voting gadget to deliver rapid, provable finality. Rewards sourced from transaction fees and, where applicable, inflationary issuance are apportioned among honest proposers and attesters, while penalties and opportunity costs disincentivise inactivity. Because consensus security is proportional to the value at stake rather than electricity expended, like in *PoW*, *PoS* drastically reduces energy consumption and lowers barriers to broad network participation.

1.3 Bitcoin Overview

Bitcoin went live in January 2009 with the *genesis block*[10]. The system introduced a public ledger secured by computation rather than by any central party. Each transfer spends and creates *unspent transaction outputs* (UTXOs); because every input must point to an existing, unspent output, the ledger forms an acyclic graph that auditors can follow to verify supply, as shown in Figure 1.7.

Time ordering relies on Proof-of-Work mining, outlined in Section 1.2.7. Miners search for a block header whose double SHA-256 digest from Section 1.2.3 falls below the current target. Nodes accept the chain with the greatest accumulated work, so an attacker would need to control nearly half of the global hash rate to rewrite history.

New coins enter circulation through the block subsidy. The reward began

[10] Block 0 contains the newspaper headline "Chancellor on brink of second bailout for banks" from *The Times* (3 Jan 2009). The line fixes the block date and alludes to the financial crisis.

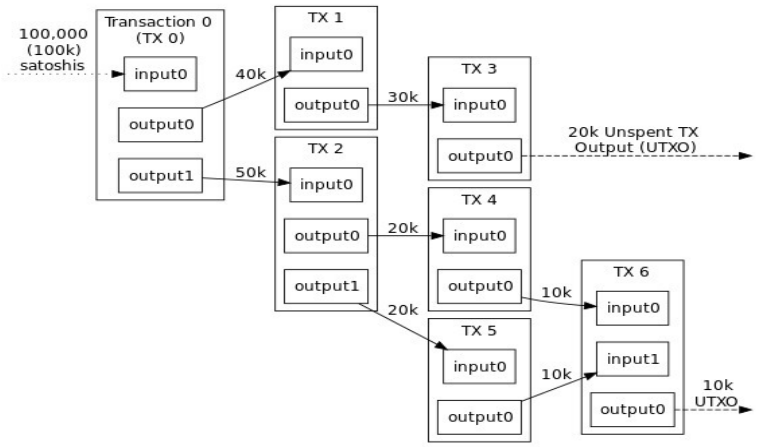

Figure 1.7: UTXOs as used by Bitcoin. [32]

at 50 BTC and drops by half every 210,000 blocks (halving [11]), placing a hard limit of 21 million BTC on total supply. The calculation for reaching max supply is shown below.

$$\sum_{i=0}^{32} 210000 \cdot \frac{50}{2^i}, dove :$$

- The variable i denotes the halving index, starting from zero and increasing up to the 32 halvings projected for the future.
- *210000* is the number of blocks that must be mined before the next halving occurs, and i is incremented.
- The value *50* is the initial block reward in BTC; this amount is divided by 2^i at the i-th halving.

Figure 1.8 illustrates how Bitcoin's total supply grows while the inflation rate steadily declines.

Bitcoin can handle about seven transactions each second because blocks arrive every ten minutes, and their size is small. Systems such as the Lightning Network move most payments off-chain and settle only the final balance on the main chain [33]. Projects like sidechains, merge-mined extensions, and covenants add extra tools around Bitcoin, but they still depend on its limited scripting rules. Privacy faces the same limits. Addresses hide names

[11] Procedure that halves by 50% the creation of new Bitcoins every 210000 mined blocks (about every 4 years considering that the mining time for the single block is 10 minutes).

(pseudonymity), but not for long because cluster analysis often links them together [34].

After more than fifteen years online, Bitcoin is still a solid reference point for open settlement networks. Directly or indirectly, the idea of Bitcoin has contributed, and continues to contribute, to the development of new innovative solutions in the Web3 landscape.

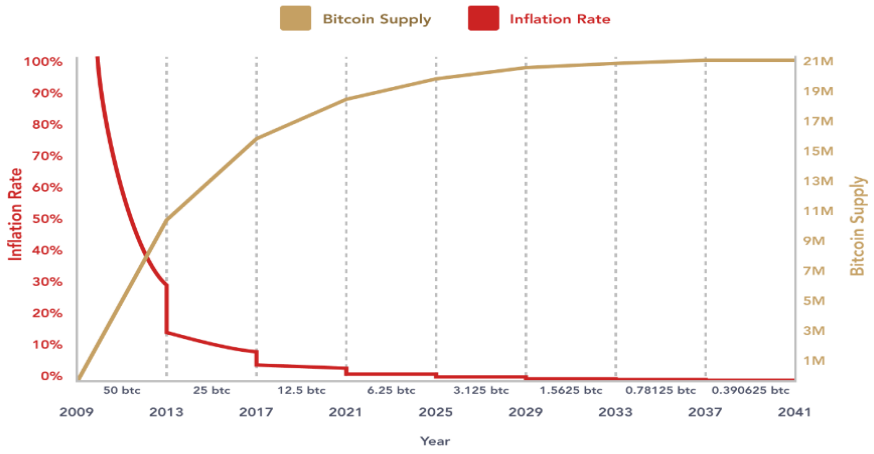

Figure 1.8: Cumulative bitcoin in circulation as block height increases.

1.4 Ethereum Overview

While Bitcoin spearheaded decentralized digital currency, Ethereum broadened these principles by providing a programmable blockchain equipped for diverse use cases. Conceived by Vitalik Buterin in 2014 [35], Ethereum introduced *smart-contracts*, which are self-executing programs embedded on the blockchain that automatically perform predefined conditions. The idea behind Ethereum is to create a large decentralized system, which goes beyond being a simple means of payment or a way to transfer money, but also a structure that can be programmed into everything from the automation of monetary exchanges to actual decentralized applications.

At the heart of Ethereum is the Ethereum Virtual Machine (EVM), a universal environment that ensures consistent execution of smart contract code across all nodes in the network. This unifying approach fosters robust security and facilitates a wide array of possibilities, ranging from finance, with decentralized finance (DeFi), to gaming and digital identities. Since introducing a Turing-complete virtual machine in 2015, Ethereum, with its EVM, has transcended simple peer-to-peer payments to become a programmable

settlement layer, serving as a foundation for advanced, decentralized ecosystems.

The next subsections will explain the key components of Ethereum that matter for a zero-knowledge exploit platform such as *zkpoex*. The focus is on the account model, the EVM execution flow, the compilation toolchain, and smart contracts.

1.4.1 Account Model: Externally Owned Accounts vs. Contract Accounts

Ethereum keeps state in a modified Merkle Patricia tree whose leaves are *accounts*. Two categories exist:

- **Externally Owned Accounts (EOA):** Identified by a public key hash, they hold a *balance* and a *nonce*. They cannot store bytecode and act only via signed transactions.
- **Contract Accounts:** They embed immutable bytecode and point to a storage trie. Code is executed when the account receives a `CALL` or `CREATE`.

Each account node stores four fields: *nonce*, *balance*, *storageRoot* and *codeHash* [36]. Figure 1.9 shows the high-level layout.

The separation of signature logic (EOAs) from execution logic (contracts) is central to exploit proofs: *zkpoex* emulates an EOA that triggers a vulnerable contract and then proves the resulting state transition without revealing the payload.

1.4.2 Transactions and Gas Accounting

Ethereum recognises several transaction formats, all wrapped in the *typed-transaction envelope* introduced by `EIP-2718`. [37] The most common types are:

1. **Legacy (type 0):** Fields are *nonce*, *gasPrice*, *gasLimit*, *to*, *value*, *data*, and the signature triple (v, r, s). [36]
2. **Access List (type 1):** Adds an *accessList* array that pre-declares storage keys to lower the dynamic gas cost. [38]
3. **Dynamic Fee (type 2):** Replaces *gasPrice* with *maxFeePerGas* and *maxPriorityFeePerGas*. The base fee is burned, and only the priority fee rewards the validator. [39]

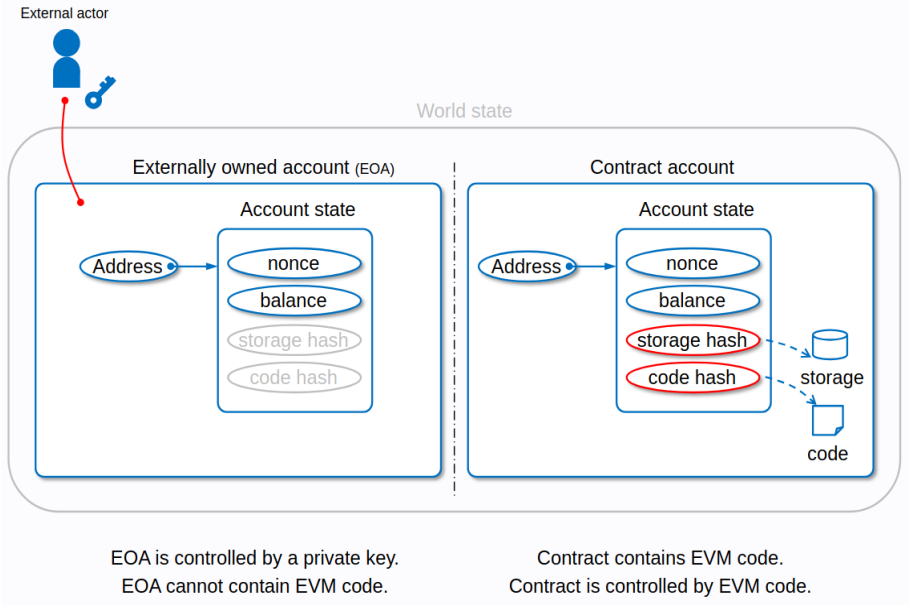

Figure 1.9: Ethereum account structure and differences between EOAs and contract accounts. [1] [2]

The prototype detailed in this thesis limits its analysis to **legacy (type 0) transactions**. By working with a single, stable encoding, the prover avoids access-list handling (for type 2 txs) and the dynamic base-fee logic of post-London blocks (for type 3 txs). This simplifies byte stream parsing inside the `Risc0` zkVM and keeps on-chain verification compact.

Construction and Broadcast

A user assembles the transaction, signs the RLP[12]-encoded payload, and submits it to the P2P network. Each node verifies the signature, checks that *nonce* matches the sender's account counter, and computes the *intrinsic gas* g_0:

$$g_0 = g_{\text{tx}} + g_{\text{data\_zero}} \times n_{\text{zero}} + g_{\text{data\_nonzero}} \times n_{\text{nonzero}} + g_{\text{accesslist}} \times n_{\text{access}}$$

as defined in Appendix G of the Yellow Paper. [36] The transaction is rejected if $g_0 > gasLimit$.

[12] RLP is a data encoding scheme used to serialize objects into a space-efficient binary format, particularly for transactions and other data structures.

Mempool and Inclusion

Once a node validates a signed payload, it places the transaction in its *mempool* a local priority queue shared with peers. Every block builder repeatedly scans this pool, sorting candidates by the *effective fee*:

$$\text{effFee} = \min\big(\mathit{maxFeePerGas},\ \text{baseFee} + \mathit{maxPriorityFeePerGas}\big),$$

then fills the next block with the most profitable bundle that fits beneath the block-gas target. The flow from wallet to on-chain inclusion is sketched in Figure 1.10. After `EIP-1559`, the single *gasPrice* field is split into two: an **algorithmic base fee**, burned by the protocol to balance demand, and an optional **tip** that rewards validators for fast service.

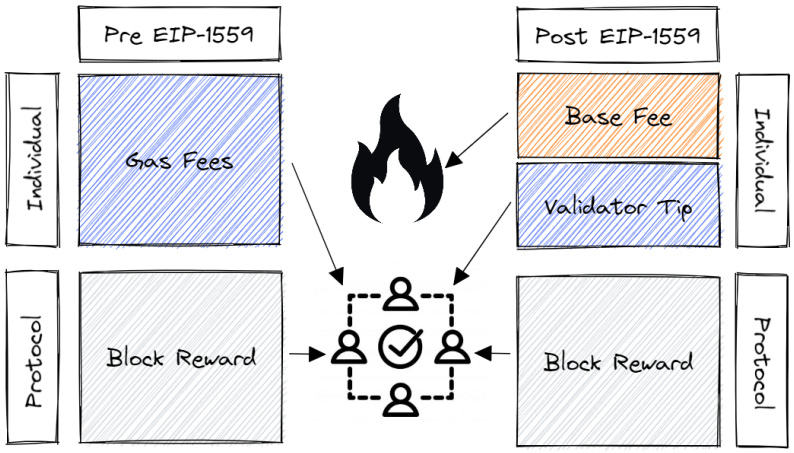

Figure 1.10: Fee mechanics before and after `EIP-1559`. [3]

Execution and Settlement

At block time, the EVM executes transactions in the chosen order, deducting *intrinsic gas* up-front and consuming the remaining budget opcode by opcode. If gas is exhausted, the call reverts, state changes in that call frame are discarded, yet the gas already spent stays with the validator. Whenever execution finishes successfully, any unused gas is refunded to the sender, subject to the usual gasRefund $\leq \frac{1}{2}$ gasUsed cap, and logs are appended to the block receipt trie. The final state root, transaction root, and receipt root are then committed in the block header, making the result immutable.

1.4.3 Smart Contracts on Ethereum

Smart contracts are self-contained programs deployed directly on the Ethereum blockchain. Once a contract's bytecode is published through a `CREATE` transaction, every full node executes the same immutable code whenever a subsequent call targets the contract address, guaranteeing that all participants observe an identical state transition. In practice, these on-chain programs behave like autonomous "digital vending machines": given a valid transaction carrying the correct function selector and arguments, they automatically transfer funds, update storage, or emit events, requiring no intermediary.

`Solidity` is the de facto language for writing such contracts and is therefore the language chosen for *all* vulnerable case studies in this thesis.

Its popularity brings three concrete advantages:

1. **Dominant footprint**: the vast majority of on-chain applications, audits, and public exploit reports are written in Solidity so that researchers can reuse known patterns without a steep learning curve.

2. **Mature tooling**: frameworks such as Hardhat, Foundry, Slither, and Echidna streamline compilation, testing, and static analysis.

3. **Direct EVM target**: the Solidity compiler outputs raw EVM bytecode, meaning an exploit written in Solidity exercises exactly the same low-level execution paths that *zkpoex* will replay and prove inside the RISC Zero zkVM.

By building on Solidity's rich ecosystem, *zkpoex* significantly improves the handling of smart contract bytecode and state transitions. Leveraging `solc`[13], developers can compile existing exploit proofs-of-concept into actual runtime bytecode, automatically extract ABI, bytecode, metadata, perform library linking, and target specific EVM versions. As a result, there is no need to manually reconstruct EVM traces or simulate low-level behavior. Instead, the exact binary generated by `solc` becomes the input to *zkpoex*, which feeds it into the zkVM for zero-knowledge proof generation.

1.5 Zero-Knowledge Proofs (ZKPs)

Zero-knowledge proofs (ZKPs) appear to be an innovative solution to ensure privacy and security in various digital interactions. By allowing a party to demonstrate knowledge of a fact without revealing any additional information, ZKPs offer a powerful tool for building trust in systems where confidentiality is paramount. This capability has significant implications in fields

[13] `solc` is the official command-line compiler for Solidity; it turns `.sol` source files into EVM bytecode and ABI description in a single step.

such as secure communications, authentication, and blockchain technologies. In the following sections, we will elaborate the key concepts underlying ZKPs, provide the historical context for their development, and explore very simple illustrative examples to clarify their mechanism. Through this exploration, we will delve deeper into how ZKPs work and their importance in modern cryptographic practices.

1.5.1 History of ZKPs

In 1985, the foundational idea of zero-knowledge proofs (ZKPs) was introduced in a peer-reviewed academic paper titled "The Knowledge Complexity of Interactive Proof Systems" [40]. This work represented a significant advancement in the field of cryptography, authored by researchers Shafi Goldwasser, Silvio Micali, and Charles Rackoff from MIT.

According to the author's description, the original idea involved "*interactive protocols*", in which the prover and verifier would repeatedly communicate in order to persuade the verifier that the prover was correct. Despite being a breakthrough in its own right, this method was time and resource-intensive, especially when dealing with large amounts of data. Zero-knowledge proofs must not be interactive in order to be *scalable*.

In 1986, Amos Fiat and Adi Shamir showed that the interaction in many public-coin zero-knowledge proofs could be eliminated by replacing the verifier's random challenge with the output of a publicly available hash function, an idea now known as the *Fiat–Shamir heuristic* [41]. Their transformation converts an identification protocol into a digital-signature scheme and, more broadly, turns an interactive proof of knowledge into a *non-interactive zero-knowledge* (NIZK) proof. Because no live exchange is required, the resulting proofs are ideally suited to settings, such as online transactions or blockchains, where prover and verifier cannot communicate in real time.

Two years later, Manuel Blum, Paul Feldman, and Silvio Micali provided the first *formal* framework for NIZK by introducing the *common reference string* (CRS) model [42]. They proved that, given a short random string shared in advance, every language in *NP* admits a computational zero-knowledge proof with no interaction, thereby establishing the theoretical generality of NIZKs beyond the random oracle setting assumed by Fiat and Shamir. Illustrative figure in 1.11.

After a few years, the next push toward ZKPs occurred in 2011, when Nir Bitansky, Ran Canetti, and Alessandro Chiesa published a paper at the International Symposium on Theory of Cryptography titled "*From extractable collision resistance to succinct non-interactive arguments of knowledge, and back again.*" [43]. This study demonstrated how to generate *SNARKS* (succinct non-interactive arguments of knowledge) using a function known as the *Extractable Collision Resistance (ECR)* hash function. In essence, SNARKS

are ZKPs that are "succinct", which means they are small in size and can be verified in a few seconds.

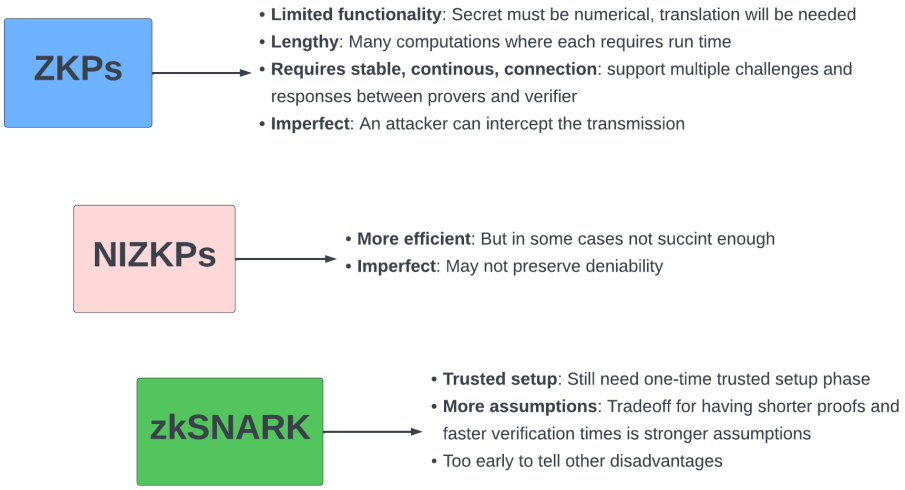

Figure 1.11: A comparison of ZKPs, NIZKPs and zk-SNARKs [4]

Despite the verification speed and scalability of zk-SNARKs, some implementations, such as *Pinocchio* [44], had some limitations:

- **Trusted setup:** Classical zk-SNARKs begin with a one-off "*ceremony*" that produces public parameters for everyone to use. The people running this ritual briefly handle secret randomness, often called *toxic waste*, which must be destroyed. If even one participant kept a copy, they could later forge convincing proofs, so the whole system inherits a small but real point of trust. An illustration can be seen in Figure 1.12.
- **Not post-quantum secure:** These proofs lean on hardness assumptions such as the elliptic-curve discrete-log problem. A large-scale quantum computer could solve that problem efficiently, undermining cryptography and allowing forged proofs.

From this moment on, the development of ZKPs accelerates greatly. Indeed, only two years later, in 2016, Jens Groth showed that zk-SNARKs could be both short and fast. His construction, now simply called *Groth16*, as shown in the paper "*On the Size of Pairing-based Non-interactive Arguments*" [45], shrinks the public parameters and the proof itself to just three curve points each. Verification is reduced to a handful of pairings that finish

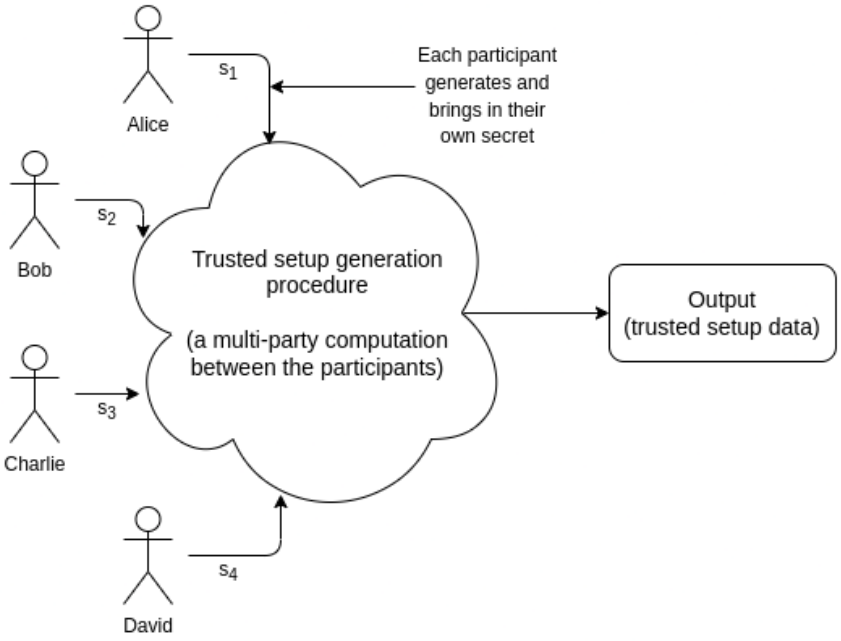

Figure 1.12: A trusted setup ceremony procedure, where s are all secrets. [5]

in milliseconds on a laptop, while the proof fits in roughly two hundred bytes. The performance leap was so large that privacy-oriented blockchains such as Zcash[14] adopted Groth16 almost immediately, and a thriving ecosystem of circuit compilers, libraries, and auditing tools has formed around it. It is still being used in a lot of protocols today, like the one on which the software of this thesis was developed: Risc0. As will be discussed in more detail in Section 2.3, Risc0 uses a "STARK-to-SNARK" circuit, and as a SNARK, it uses precisely *Groth16*.

The next wave arrived in 2017 with *Bulletproofs* by Bünz, Bootle *et al.*, [46]. Bulletproofs are logarithmic-length, non-interactive proofs that rely only on the discrete log assumption, so they dispense entirely with the trusted setup. Their flagship feature is the *range proof*: one can convince a verifier that an encrypted value falls inside a chosen interval without revealing the value itself. Many such statements can be aggregated into a single short proof, keeping confidential transactions on Bitcoin-style chains small enough for everyday use. The absence of toxic waste, combined with good practical performance, made Bulletproofs an instant favourite in "trustless" blockchain design.

[14] Cryptocurrency that focuses on privacy and anonymity, offering users the option of both transparent and shielded transactions.

Together, Groth16 and Bulletproofs reshaped the field: the former proved that zk-SNARKs could be truly lightweight, while the latter showed that trusted ceremonies are not a necessity. Their success paved the way for the flurry of post-2018 research into transparent, post-quantum-ready proof systems and the recursive techniques that power today's layer 2[15] scalability solutions. The spotlight then shifted in 2018 when Ben Sasson *et al.* introduced zk-STARKs, *Zero Knowledge Scalable Transparent ARguments of Knowledge*, which swap pairing based cryptography for hash based commitments and replace the one off ceremony with a public randomness beacon, making the system both post quantum resilient and fully transparent [47].

Although the resulting proofs are noticeably larger, often running to tens or even hundreds of kilobytes, the design proved that trusted setup is not a law of nature and sparked a race to shrink proof sizes without reintroducing toxic waste. PLONK[16] followed in 2019: by adopting a universal, updatable structured reference string, it slashes the number of ceremonies to one, and that ceremony can be refreshed at will, thus reducing the trust footprint while sustaining Groth16 level efficiency [48]. In parallel, Bowe, Grigg, and Hopwood unveiled *Halo*, showing that recursive composition is possible in the discrete log world without any setup at all; their technique allows unlimited proof aggregation and laid the groundwork for succinct block verification [49]. Recursion soon became leaner still: Setty's *Nova* (2021) introduces a folding scheme whose verifier costs only a handful of group operations and whose prover needs no FFTs[17] or ceremony [50]. The fold and aggregate idea has since been generalised by protocols such as *Protostar* and *HyperNova* (2023), which support high-degree gates, lookups, and other features required by modern zk virtual machines [51]. Industry deployments kept pace: Polygon's *Plonky2* (2022) marries STARK-style arithmetisation with PLONK-inspired recursion, producing millisecond-scale proofs on consumer hardware and powering roll-ups like Polygon zkEVM [52]. By 2024/25, the ecosystem has converged on hybrid stacks, fast STARK proofs compressed by compact SNARK wrappers, running on GPU- and VPU-accelerated hardware; a recent survey lists more than twenty open-source frameworks that developers can benchmark out-of-the-box [53]. Building on these hybrid foundations, in May 2025, StarkWare unveiled *Stwo*, a next-generation open-source STARK prover written in Rust that implements the new Circle STARK protocol [54]. Early benchmarks exceed 500 k-hash/s

[15] Second-layer (off-chain or partially off-chain) scalability protocols that periodically commit their state to the main blockchain.

[16] Polynomial Commitment scheme over Lagrange bases for Non-interactive arguments of Knowledge.

[17] Fast Fourier Transforms: algorithms that compute discrete Fourier transforms in $O(n \log n)$ time and are heavily used in polynomial-based proof systems.

on a commodity quad-core laptop[18], and a WebAssembly build has already demonstrated full client-side proving directly in the browser: the first time complete proofs can be generated and verified on off-the-shelf devices. *Stwo* combines aggressive arithmetic optimisations with a modular design slated for Starknet's production stack, paving the way for web-scale, on-device proof generation. In less than a decade, zero-knowledge proofs have evolved from megabyte-scale, ceremony-bound prototypes to sub-kilobyte, post-quantum-ready proofs that settle blockchain state every few seconds, and the trajectory suggests that constant-time, constant-memory ZKPs are now within realistic reach of 2030-era devices.

1.5.2 Illustrative examples of ZKPs

ZKPs are often a very complicated concept to understand, so to visualize this idea easily, we will present two famous abstract examples.

The Ali Baba Cave

The first example we will discuss was first exposed in [55], it offers a particular insight into how protocols using zero-knowledge proofs operate. Picture Ali Baba, an elderly merchant from Baghdad, who visits the bazaar daily to trade goods. One day, a thief grabbed a purse from Ali Baba and fled into a uniquely shaped cave. Upon reaching the cave, Ali Baba encountered two dark, winding passages: one to the left and the other to the right. Having missed which way the thief went, he was compelled to make a choice. After a moment of contemplation, he decided to investigate the right passage. Unfortunately, he found no sign of the thief, only to reach a dead end. Concluding that the thief might have taken the other passage, he ventured left, but again found nothing; that route also led to a dead end. Disheartened, Ali Baba realized the thief had likely escaped while he was still in the first passage. Sad about his loss, he returned home and went to sleep. The next day, as Ali Baba made his way back to the bazaar, another thief grabbed Ali Baba's basket and headed for the same cave. Once more, Ali Baba faced the same dilemma.

As the days passed, Ali Baba found himself repeatedly robbed by a thief who always escaped into the cave, leaving him frustrated and confused. Each time, he convinced himself that luck favored the thieves, allowing them to choose the passage he hadn't selected. However, on the fortieth day, after being robbed for the fortieth time, Ali Baba began to suspect that the cave

[18] "Hash/s" counts the Keccak/Fri-hash evaluations that dominate STARK proving time. At 500 000 hash/s, a 2 M-hash roll-up circuit completes in roughly four seconds; first-generation STARK prototypes (in 2018) achieved below 5 000 hash/s on similar hardware, pushing the same circuit well past the ten-minute mark.

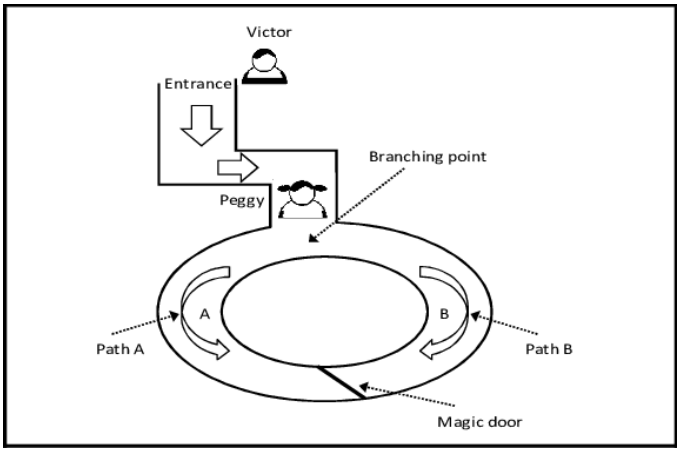

Figure 1.13: Ali Baba's cave representation.

concealed a significant secret. Determined to uncover the truth, he decided to hide in one of the dark passages and wait for a thief to reach a dead end. Hours went by, and finally, a thief rushed through the passage. Spotting another merchant behind him, the thief uttered the magical words, "Open sesame." To Ali Baba's surprise, a hidden entrance in the wall of the dead end slid open, allowing the thief to slip through and close the passage behind him. Ali Baba realized that the right passage was connected to the left one when the wall moved. Intrigued, he spent a long time experimenting with the magic words, eventually modifying them as if he were changing the code on a lock. The next day, when a thief attempted to rob him again, Ali Baba was ready and managed to trap him, preventing any escape. He recorded this experience in an illustrated manuscript, but he deliberately omitted the new magic words, leaving behind subtle hints for anyone clever enough to discover them. Years later, a researcher stumbled upon the manuscript and deciphered the magic words. Excited by this revelation, he invited a group of reporters to the cave to witness his discovery, but he cleverly withheld the actual magic words from them. How did he manage this? The process was straightforward. First, a television crew filmed an extensive tour of the cave, capturing the two dead ends, before leaving. The researcher then re-entered the cave through one of the passages. Following him, the reporter, with a camera in tow, entered and reached the fork in the path, where he flipped a coin.

- If the coin landed heads, he would instruct the researcher to exit from the left.
- If it landed tails, he would signal for the researcher to come out from the right.

The researcher would then emerge from whichever direction the reporter indicated. This procedure was repeated 40 times, commemorating the same number of times Ali Baba had been robbed. Each repetition cut the probability of the researcher emerging from the correct passage in half, leading observers to believe he must have known the secret words after consistently appearing from the designated exit. Now we arrive at the intriguing part of the tale. After the original tape was recorded, another reporter sought to tell this remarkable story, but the researcher refused to cooperate. In a twist of mischief, he instead disclosed the cave's location to the envious reporter. This reporter hired an actor resembling the researcher and returned to the cave to replicate the earlier recording. However, this time the actor was unaware of the magic words. Each time he exited the wrong passage, the reporter edited the footage, cutting those segments and repeating the process until the coin flip aligned with the actor's exit. After numerous edits, they produced a tape featuring 40 successful trials, although the process took far more attempts than that. As the two rival networks aired their contrasting stories simultaneously, the issue escalated to court. Judges and experts found themselves unable to distinguish between the tapes, making it impossible to ascertain which was the authentic recording. The simulation revealed no knowledge of the secret, as even the actor remained oblivious to the magic words. Consequently, the original tape and the simulation were indistinguishable; therefore, the genuine recording also failed to convey any knowledge of the secret. The reporter who had witnessed the real researcher, rather than the actor, emerge correctly from the chosen passage each time was convinced that the researcher possessed the secret words. Yet, he was unable to communicate this certainty to the judge or the experts. Ultimately, the researcher had achieved his true aim: demonstrating that one could persuade others without revealing the secret, thus keeping the magic words undisclosed.

Where's Wally?

The second and final example we will cover is maybe the simplest way to prove that you have knowledge of something without giving it away, which can be shown with the often-used "Where's Wally?" (called "Where's Waldo?" in North America) example. This idea was originally written about in a paper by M. Naor, Y. Naor, and Reingold. [55].

Let's imagine Alice and Bob are searching for an imaginary character named Wally in an image together. Alice knows where Wally is in the image, but Bob does not believe her. How can Alice prove to Bob that she knows Wally's location without revealing it? Imagine Alice has a large sheet of paper that covers the entire image, with a small cutout through which Bob can see Wally. By using this method, Alice can demonstrate to Bob

Figure 1.14: Bob's view representation.

that she knows Wally's location without revealing Wally's exact position on the image (A visual demonstration of what Bob sees after Alice's action, in figure 1.14. Bob can view Wally only from the hole.). This scenario serves as a straightforward example of a non-interactive zero-knowledge proof. The observer, seeing Wally through the cutout, is assured that Wally is present and that Alice knows his location, without disclosing any additional details. Note that this is not a perfect ZKP, as Alice has revealed some information about Wally, such as his body position and clothing.

1.5.3 From intuition to formalism

The illustrative examples in 1.5.2 show *why* secrecy can coexist with convincing evidence; this section pinpoints the *how*. Five building blocks: interactive proofs, witnesses, commitment schemes, proofs of knowledge, and zero-knowledge flavours form the common grammar behind today's ZK systems, whether interactive, Fiat–Shamir, or CRS-based.

Interactive proof systems

An interactive proof is a finite conversation between a prover $\mathcal{P}$ and a verifier $\mathcal{V}$. After several challenge–response rounds, $\mathcal{V}$ outputs accept or reject. Every language in PSPACE admits such a protocol. Security rests on three pillars:

- **Completeness**: an honest prover convinces an honest verifier except with negligible error;

- **Soundness**: a cheating prover persuades the verifier only with probability $< 2^{-\lambda}$ for a public parameter λ;
- **Zero-knowledge**: for every verifier there exists an efficient simulator whose transcript is indistinguishable from a real one, so nothing beyond the statement's truth leaks.

These guarantees formalise the tape-swap experiment recalled in 1.5.2.

Instances, relations and witnesses

A statement is an *instance* x. It lies in a language L_R if there exists auxiliary data w (the *witness*) such that $(x, w) \in R$ for a polynomial-time relation R. Square roots certify quadratic residues; secret exponents certify discrete-log instances. In practice, x is public input while w stays private.

Commitment schemes

Many protocols begin with the prover *freezing* a value that will be opened later. A commitment scheme offers two promises: *binding* the sender cannot change the committed message, and *hiding* the receiver learns nothing until the opening is revealed. Hash-and-concatenate designs give perfect binding and computational hiding; Pedersen commitments invert that trade-off under the discrete-log assumption and support homomorphic addition.

Proofs of knowledge

Sometimes the verifier needs proof that the prover actually *possesses* the witness. A protocol is a proof of knowledge if a polynomial-time *extractor* can rewind any successful prover and output a valid witness with comparable probability. This strengthens soundness into *knowledge soundness*.

Zero-knowledge flavours and compilation

Privacy comes in three flavours:

- **Perfect ZK**: the simulator's and real transcript distributions are identical;
- **Statistical ZK**: the two differ by at most $2^{-\lambda}$; STARKs achieve this with random masks and FRI sampling;
- **Computational ZK**: indistinguishable only for polynomial-time adversaries; classic SNARKs fall here.

Interaction can be removed via the Fiat–Shamir transform, which hashes the transcript into a challenge, or via a Common Reference String (CRS) sampled once in advance for all proofs. Both preserve completeness, soundness, and the chosen flavour of zero knowledge under suitable assumptions.

With these five concepts: interactive proofs, witnesses, commitment schemes, proofs of knowledge, and privacy flavours, we have a concise toolkit against which every later protocol in this thesis will be measured.

1.6 Zero-Knowledge Virtual Machines (zkVMs)

Zero-knowledge virtual machines elevate the privacy guarantees discussed in Section 1.5 from hand-written circuits to *general-purpose computing*. A zkVM allows a prover to run ordinary machine code on private data, record the entire execution trace, then compress that trace into a succinct proof that every instruction respected the architecture's transition rules. The verifier re-executes nothing; it checks only the cryptographic commitment plus a small public journal. In *zkpoex*, the whole Rust-based EVM simulation is executed inside the RISC Zero zkVM. It will be discussed in more detail in the next chapters.

1.6.1 Concept and Definition

A zkVM behaves like a virtual CPU equipped with a mathematical shadow: every opcode of the guest ISA[19] is rewritten as algebraic constraints over a finite field. During proving, a local executor interprets the compiled binary exactly as a conventional emulator would, but it also logs the register file, memory bus and control-flow decisions at every step. Those rows form a two-dimensional trace table that is cryptographically committed with a STARK, a SNARK, or a hybrid thereof. Verification amounts to checking a handful of low-degree polynomial identities against that commitment; soundness relies on the collision resistance of the commitment scheme, whereas zero-knowledge is obtained by masking witness values with fresh randomness. As summarised in Figure 1.15, the artefacts produced at each stage flow deterministically from compiler to verifier.

The information paths visible in Figure 1.15 can be dissected into four conceptual blocks:

- **Compiler stage:** Source code written in C, C++, Rust, or Solidity is translated into machine code whose exact binary encoding is dictated by the chosen ISA. Any optimiser run at this level is untrusted; the

[19] Instruction-set architecture (ISA): the contract between software and hardware that enumerates instructions, registers, and observable side effects.

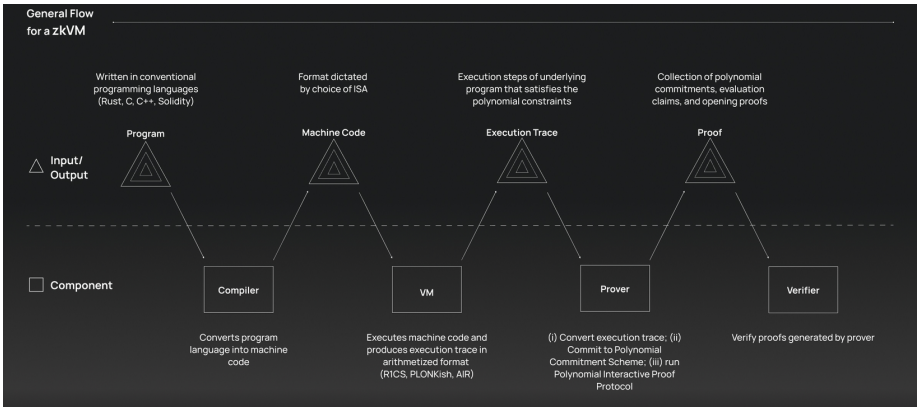

Figure 1.15: Abstract workflow of a zkVM: compilation and local execution (left), trace commitment (centre), and verification (right).

later stages will still catch mis-compilations because the proof must replay every instruction [56].

- **VM execution stage:** The zkVM executes the machine code and emits a complete *execution trace*, a row for each clock tick. The column layout of this table is predetermined by the arithmetisation scheme: R1CS matrices for Groth16, PLONK-ish traces for Halo 2, or AIR tables for plonky2 and RISC Zero [57].

- **Prover stage:**

  (i) The trace is interpolated into a set of polynomials subject to the ISA constraints.

  (ii) These polynomials are bound with a polynomial commitment scheme (PCS), yielding a short fingerprint that hides all private rows.

  (iii) A Polynomial Interactive Oracle Proof (PIOP) is executed in the Fiat–Shamir model, turning the interaction with the verifier into non-interactive challenge hashes.

  (iv) Finally, the prover outputs an *opening proof* that convinces any verifier that the blinded evaluations are consistent with the earlier commitment [58].

- **Verifier stage**. Given the seal, the verifier reconstructs the same challenges, checks the opening proof against the commitment, and either accepts or rejects. Thanks to algebraic batching, only a handful of group or field operations are required, enabling cheap on-chain verification.

A zkVM proof, therefore, certifies that for a *specific* program, and for *some* (undisclosed) input, the recorded execution necessarily produces the claimed public output when run on the stated ISA, without revealing anything else about the private data processed along the way.

1.6.2 Historical Evolution of zkVMs

The trajectory of zero-knowledge virtual machines traces back to the first privacy-preserving roll-ups of 2017–2019, when developers embedded highly specialised circuits, token transfers, Merkle proofs, and mixer arithmetic directly into proving systems. Such vertical integration proved brittle: every new feature demanded a fresh circuit and weeks of re-auditing. The landscape shifted in 2020 when *Cairo VM* introduced a STARK-friendly, Turing-complete instruction set that powered StarkEx's production roll-ups the very same year, [59, 60]. Polygon's announcement of *Miden VM* in 2021 demonstrated that a stack-based ISA, carefully designed for low-degree constraints, could achieve transparent proofs without a trusted setup. In November 2021, the alpha release of Starknet confirmed that general-purpose contracts could run under zero knowledge without bespoke circuits [61]. Early 2022 brought the open-source debut of *RISC Zero*, which executes unmodified 32-bit RISC-V binaries, stitches STARK segments recursively, and finally wraps them with Groth16 to compress the seal to roughly one kilobyte [62] (this is the proof system used for *zkpoex* as explained more in depth in section 2.3). The launch of *zkSync Era* in March 2023 added a byte-accurate zkEVM aimed at mainstream Solidity developers [63]. Succinct's *SP1* (2024) then layered Nova-style folding on a custom trace format, reaching laptop-grade proving throughput while remaining fully open-source [64]. By 2024, the community gradually distinguished *zkVMs* (language-agnostic, general-purpose) from *zkEVMs* (EVM-specific) to clarify design trade-offs:

- **zkEVMs** prioritize Ethereum compatibility, enabling seamless porting of Solidity contracts but inheriting the EVM's inefficiencies (e.g., stack-based architecture, high opcode complexity) [63]. Projects like zkSync and Polygon zkEVM fall into this category, optimizing for developer familiarity [65].
- **zkVMs** (e.g., RISC Zero, SP1) favor flexibility, supporting arbitrary binaries (Rust, C++) and enabling novel use cases like *zkpoex*. They avoid EVM constraints, often achieving better proving efficiency [64].

For *zkpoex*, a zkVM was chosen over a zkEVM for three reasons:

1. **Performance**: The Rust-based EVM emulator (Rust-based Ethereum Virtual Machine) ran more efficiently on RISC Zero's RISC-V ISA than on an EVM-equivalent architecture [56].

2. **Generality**: *zkpoex* required custom constraints (e.g., reentrancy detection) that were easier to implement in a zkVM's flexible proving environment [57, 58].

3. **Toolchain Maturity**: RISC Zero's Rust SDK and composable proof system (STARK→SNARK) simplified integration with existing security tooling [62].

This divergence reflects a broader trend: zkEVMs dominate L2 scaling, while zkVMs enable applications beyond Ethereum.

1.6.3 Evaluation Criteria for zkVMs

When comparing zero-knowledge virtual machines, it is tempting to look only at prover speed, yet practical deployments reveal a richer set of trade-offs. It's interesting how the Lita Team[20] defined a holistic approach to evaluating zkVMs. They group the most relevant properties into two families, *baseline reliability* and *performance*, and argue that both must be inspected before declaring one design "better" than another.

Baseline reliability: These attributes determine whether a zkVM can be trusted at all.

- **Correctness.** The virtual machine must execute the guest program exactly as specified, and the proof system must uphold its three pillars: *soundness* (false statements cannot be proven), *completeness* (every true statement is in principle provable), and *zero-knowledge* (the proof leaks no information beyond the claimed public output). Failing any pillar turns the system into a liability rather than an asset.

- **Security level.** Each pillar carries a statistical bound. Practice accepts a failure probability of at most 2^{-n} for some security parameter n; higher n yields stronger guarantees but increases proving effort. Modern STARK-to-SNARK hybrids aim for $n \geq 96$ bits as a sensible minimum.

- **Trust assumptions.** Proof systems fall into three buckets. Designs with *no trusted setup* rely only on public randomness. Systems with a *one-out-of-(N)* assumption require at least one honest participant in a multi-party ceremony. Block-production schemes may add an *honest majority* assumption. All else equal, fewer assumptions imply a more robust zkVM.

[20] The Lita Team is a team of computer scientists and applied cryptographers dedicated to building the verifiable computer of the future.

Performance metrics: Once baseline reliability is satisfied, attention shifts to resource economics.

- **Efficiency.** Core-time per trace row matters for energy-sensitive workloads that must keep proving costs predictable and low.
- **Throughput (speed).** Latency-sensitive applications such as high-frequency trading prefer raw wall-clock speed even at the cost of higher energy consumption.
- **Succinctness.** Proof size and on-chain verification cost dominate user fees for public blockchains. Pairing-based wrappers typically shrink seals to $\approx 1\,\mathrm{kB}$ and two pairings, whereas fully transparent STARKs remain in the tens of kilobytes.

In practice, the ideal balance depends on the concrete use case. Price-sensitive archival systems may favour core-time efficiency, while DeFi protocols gravitate toward minimal latency. Developers must therefore benchmark candidate zkVMs across *all six* axes before committing critical workloads.

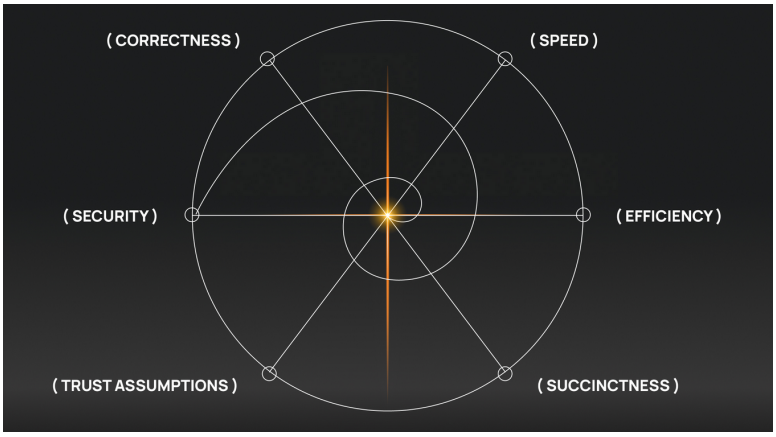

Figure 1.16: Radar chart of the six evaluation axes: correctness, speed, efficiency, succinctness, trust assumptions, and security; used to benchmark a zkVM. [6]

1.6.4 The zkVM Trilemma

Designers confront three competing objectives:

1. *Prover throughput*: hash-based STARK commitments parallelise well on GPUs but emit proofs measuring tens of kilobytes;

2. *Proof size and verification cost*: pairing-based SNARK wrappers shrink the proof to ≈ 1 kB and verify with two pairings, yet add cryptographic overhead at prove-time;

3. *Guest compatibility*: byte-accurate zkEVMs reuse existing tooling but inherit an awkward gas-charging microcode, whereas bespoke ISAs simplify constraint systems but require developers to recompile, or even rewrite their contracts.

No implementation hits all three vertices at once: RISC Zero favours small seals and broad language support at the expense of higher prover cycles, while Cairo accepts larger proofs in exchange for maximal throughput. The trade-off between these three objectives is well summarized in Figure 1.17.

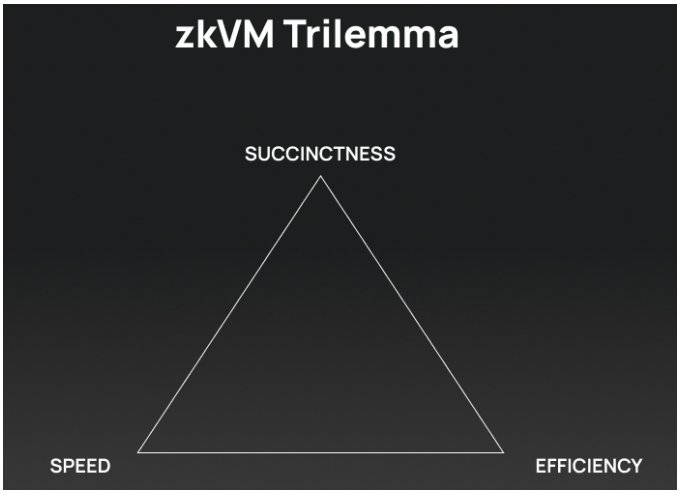

Figure 1.17: The zkVM design trilemma: trade-offs between prover speed, proof succinctness, and guest-level compatibility.

Chapter 2

Proposed System Architecture

This chapter provides a comprehensive blueprint of the *zkpoex* framework. We begin with a concise overview of the `research methodology` employed to survey relevant literature and distil the design requirements. The discussion then introduces the RISC Zero zero-knowledge virtual machine (zkVM) that underpins succinct proof generation, followed by an examination of the Rust-based Ethereum Virtual Machine (Rust EVM) and the interoperability layer that bridges these two environments. The chapter concludes with a detailed walkthrough of the exploit-execution pipeline, illustrating how each component cooperates to produce a verifiable receipt. After reading all of this chapter, you can view a schematic of the workflow of the first release of *zkpoex* (`v0.1.0`) in Figure 2.1.

2.1 Overview of the System

The *zkpoex* project couples a conventional host process with a guest program that runs inside a RISC Zero zero-knowledge virtual machine (zkVM). In addition, a Rust Ethereum Virtual Machine (Rust EVM) interpreter is launched in the guest program to simulate smart-contract attacks. The host supplies public context: chain height, gas price, account state hashes, together with confidential artefacts such as the exploiter bytecode and crafted calldata. The guest, compiled to RISC-V, replays a single Ethereum transaction inside the Rust EVM and emits a receipt through the *zkpoex* prover whose journal binds the initial public state, the post-execution state root, and a commitment to every hidden input.

This separation lets an auditor (or private projects) convince a verifier that, given a *calldata*, to prove an exploit, he must first define a set of conditions called *program_specifications*. *zkpoex* uses this specification to verify whether a state transition invalidates any conditions in the smart contract. You can read a formal explanation of this in section 2.

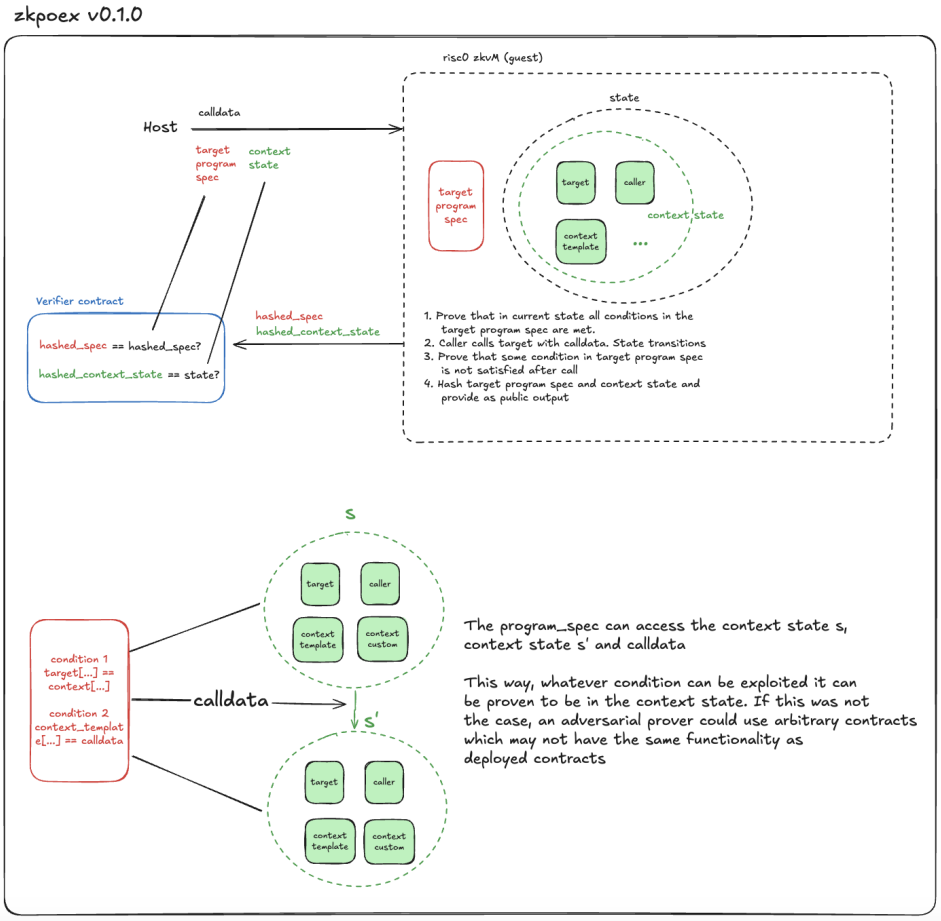

Figure 2.1: A schematic of the *zkpoex* workflow [7]

At a higher level of abstraction, the *zkpoex* system can be viewed as two principal components: The *zkpoex* Prover and The Smart-Contract Owner. Before delving into these main components, we will explore some key concepts introduced in `zkpoex`:

- Context State,
- Program Specifications.

2.1.1 Context State

The `context_state` is a compact snapshot of the on-chain world that the prover is allowed to touch. Each entry in the top-level array represents one Ethereum account and is encoded as a plain JSON object:

- **address**: 20 byte address, left-padded to 40 hex characters; serves as the Merkle-Patricia key.
- **nonce**: transaction counter, stored as a decimal string to avoid surprises when serialising.
- **balance**: Wei balance, again string-encoded so that very large values survive round-trips.
- **storage**: sparse map `slot → value` where both keys and values are 32-byte hex words; only touched slots are listed, which keeps the witness small.
- **code**: run-time bytecode of the contract, hex-prefixed; an empty string marks an EOA.

During proof generation, the guest computes the Keccak256 Merkle root of this structure and exposes it in the journal. Because the contract owner supplies the state and becomes public, the prover cannot smuggle in a `favourable` pre-state; at the same time, limiting the snapshot only to the accounts of interest keeps the circuit lean and the proof time acceptable.

2.1.2 Program Specifications

While `context_state` is a simple, enumerable snapshot of the world, a `program_specification` is closer to a `behavioural contract`. It captures, in machine-readable form, the invariants that `must` continue to hold for a contract, no matter which calldata the prover chooses. Because it encodes logical properties rather than raw data, the file is necessarily more articulated; the subsection is therefore slightly longer.

A specification is an `array of method descriptors`. Each descriptor binds one function selector to a set of post-execution predicates and an (optional) ABI schema:

- **method_id:** Four-byte selector keccak(signature)[0..4]. Only EVM calls whose calldata starts with this prefix are analysed by the zkVM.
- **conditions:** An ordered list of predicates that must evaluate to TRUE `after` the call returns. Predicates are written in a DSL whose atomic form is

  ```
  { "Fixed": { "k_s": "<path>", "op": "<Op>", "v": "<value>" } }
  ```

  where:

  - `k_s` selects either `<addr>.balance` or `<addr>.storage.<slot>`;
  - `op` is one of `Eq`, `Ne`, `Gt`, `Ge`, `Lt`, `Le`;
  - `v` is a constant against which the post-state is compared.

  The zkVM walks the list; the `first` violated predicate is enough to declare an exploit and halt the trace.

- **arguments:** ABI-typed parameter list. It exists solely so that front-end tooling and fuzzers can craft valid calldata; the verifier contract does not inspect this field.

In practice, the contract owner publishes the current bytecode, its `context_state`, and the `program_specification` on-chain, locks a bounty, for instance, and waits. A white-hat feeds the same artefacts to *zkpoex*, searches for calldata that breaks at least one predicate, and produces a zero-knowledge proof whose journal commits to the `hash` of the specification, never to the exploit input itself. If the verifier contract confirms that the committed hash matches the canonical specification, funds are released, but the calldata remains private.

Formal taxonomy of conditions

From [7]: Let $A = 2^{160}$ be the address space and $V = 2^{256}$ the word space. A `state` is a map $s : A \to V$.

Fixed condition.

$$F_s(a,\, op,\, v) \;:=\; \big(s(a)\; op\; v\big), \quad a \in A,\; v \in V.$$

Relative condition. Given a pre-state s and a post-state s':

$$R_{s,s'}(a,\, op,\, a') \;:=\; \big(s(a)\; op\; s'(a')\big), \quad a, a' \in A.$$

Specification. Let $m(i, s)$ be the state transition induced by a method m on input i. A specification is a finite set

$$S = \{(m, C_m)\},$$

where C_m is a list of conditions attached to m. S is `valid` iff for every m, i and for every pre-state s the post-state $s' = m(i, s)$ satisfies all the associated conditions:

$$\forall (m, C_m) \in S,\; \forall i,\; \forall c \in C_m,\; c(s, s') = \textsc{true}.$$

An exploit is a witness i_E such that some m turns a valid state into a state where at least one $c \in C_m$ is FALSE. Producing a *zk-STARK-to-SNARK* (you can see section 2.3 to deep in the RISC Zero's full proving stack) proof that attests this fact, without revealing i_E, fulfils the bounty and enables rapid, privacy-preserving incident response.

2.1.3 The *zkpoex* Prover

Conceptually, the *zkpoex* prover reuses the standard RISC Zero prover discussed in RISC Zero zero-knowledge virtual machine, but tailors the workflow to an Ethereum-specific payload. At a high level, the prover finds an exploit in the owner's smart contract and generates a zero-knowledge proof of the exploit execution. To do so, it shows that it knows some calldata that can be used to call the contract such that it breaks the contract's *program_specification*. The prover can verify said proof and claim rewards automatically without the need for a third party. On-chain operations and all the technicalities will be discussed in more detail in Implementation and Validation Chapter.

2.1.4 The Smart-Contract Owner

The *Smart-Contract Owner* in the *zkpoex* vision is the actor that is responsible for:

- Deploying the *VerifierContract*;
- Setting up the *program_specification*;
- Defining the rewards in the *VerifierContract* for the prover.

As mentioned in the first chapter of this thesis (1.), the *zkpoex* project wants to verify, in a totally trustless environment, that a smart-contract exploit exists without any leak of exploit details. To do so, the *Smart-Contract Owner* needs to deploy the *VerifierContract* of *zkpoex*, which interacts, depending on the Ethereum network you are using, with the deployed Verifier of RISC Zero [66].

In our setting, the guest steps through an EVM interpreter, executes the target smart contract, and checks the run against a predetermined specification. Once the computation finishes and the zero-knowledge proof is produced, two public hashes are published:

- Keccak256(S), where S is the *program_specification* in force;
- Keccak256(C), where C is the hash of *context_state* data (such as other deployed contracts).

A sound proof alone is not sufficient. To ensure the guest really inspected the contract that is live on chain and did so under the current rules, the verifier recomputes each hash from today's artefacts and compares. If the freshly derived hash of the specification differs from the advertised Keccak256(S), or the hash of *context_state* from Keccak256(C), the receipt is rejected. Such a discrepancy would signal that the proof targets outdated code or an obsolete specification. Only when every comparison coincides does the verifier accept the claim, confirming that the evidence truly pertains to the contract and specification currently deployed. After that, the *VerifierContract* sends automatically the reward to the prover.

2.2 The RISC Zero zkVM

The `RISC Zero` zero-knowledge virtual machine (you can find an explanation of what a zkVM is, in general, in section 1.6) functions on a fundamental principle: a *guest program*, utilizing a designated instruction set, is performed within a virtual machine environment, producing the appropriate proof.

As said in RISC Zero Docs: "The RISC Zero zkVM lets you prove correct execution of arbitrary Rust code. By allowing users to build zero-knowledge applications that leverage existing Rust packages, the RISC Zero zkVM makes it quick and easy to build powerful verifiable software applications." [8]

In Figure 2.2, you can see the components of a RISC Zero zkVM:

1. **Guest Code**: the Rust routine whose execution will later be attested;
2. **Executor**: runs the compiled binary and records the full execution trace;
3. **Prover**: checks that trace step-by-step and compresses it into a proof;
4. **Receipt**: the compact artefact that bundles the public journal and its zero-knowledge seal;
5. **Host Code**: the machine that is running the zkVM.

2.2.1 The Receipt

As noted in [67], a *Receipt* (shown as a gold document in Figure 2.2) is the artefact that the zkVM returns once your guest program finishes: it combines the programme's public outputs, collected in the *journal*, with a zero-knowledge proof called the *seal*. Because the seal is a succinct validity

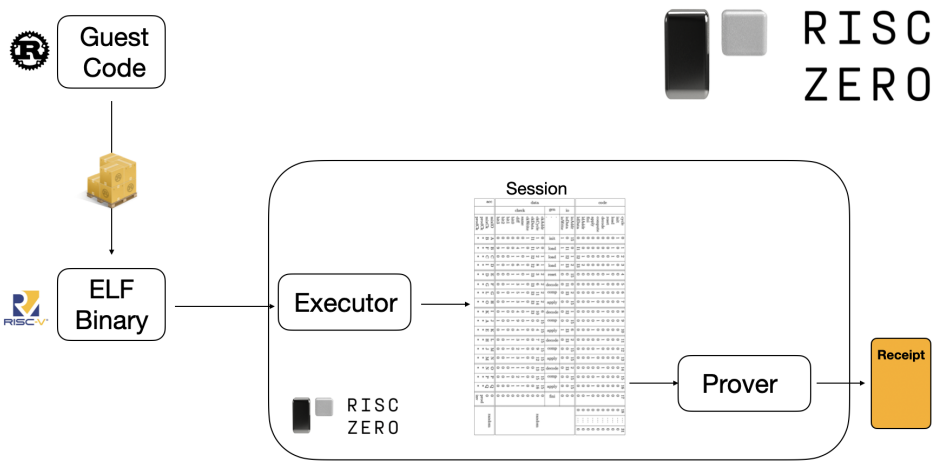

Figure 2.2: End-to-end proving pipeline in RISC Zero [8]

proof, anyone can verify without re-executing the code that the journal was produced by the expected image ID and that every step of execution was sound.

A typical workflow is therefore straightforward.

1. Alice runs the guest, producing a receipt.
2. She forwards that receipt to Bob.
3. Bob first reads the public results through `receipt.journal`;
4. Bob then calls `receipt.verify()`, gaining cryptographic assurance that:
   - the computation was carried out faithfully;
   - the guest binary matched the agreed image ID.

Receipts are ordinary Rust structures, but they can be serialised with any `serde`[21] compatible format.

2.2.2 The Guest Code

The *Guest Code* (shown with the Rust logo in Figure 2.2) is the self-contained Rust routine that the zkVM will actually execute and prove. Once compiled to a RISC-V *ELF binary* (indicated by the RISC-V logo in Figure 2.2), its initial memory image is hashed to a unique `Image ID`. That identifier

[21] A generic serialization/deserialization framework for Rust

later lets any verifier check that a receipt truly originated from the expected binary; no source code is strictly required, only the ELF (or its Image ID) needs to be shared.

But why does this determinism matter? Because every byte of the ELF contributes (via the memory image) to the Image ID, even seemingly innocuous differences like timestamps, debug sections, or compiler versions would change the hash and break verification.

2.2.3 The Executor

The *Executor* is the portion of the zkVM responsible for generating the *execution trace*. The *Executor*, basically, runs the program inside the RISC Zero environment and records a full execution *Session* (or execution trace), displayed as a tabular trace of instructions, registers, and memory activity. From [68]: "Each row describes a complete snapshot of the state of the zkVM at a given moment in time. The width of the execution trace relates to the number of registers/components in the machine, and the length of the execution trace relates to the number of clock cycles[22] of the program's execution". The *Prover* will check the validity of that recorded *Session* after.

2.2.4 The Prover

The *Prover* is the portion of the zkVM that executes and proves a *guest* program, thereby constructing a *receipt*. As shown in Figure 2.2, the *Prover* takes the recorded *Session* and checks its validity. A valid trace means that the *ELF binary* was faithfully executed according to the rules of the RISC-V instruction set architecture. In RISC Zero, there are two types of *Provers*:

- **Local Prover:** Generates proofs locally.
- **Remote Prover:** Uses a remote service called *Bonsai* [69].

2.2.5 The Host Code

From [70], the host is an untrusted agent that sets up the zkVM environment and handles inputs/outputs during execution. Although not seen in Figure 2.2, it is still an important component of a RISC Zero zkVM application. Before any proving begins, the host first prepares a dedicated execution environment for the *Executor*. This sandbox collects every run-time parameter configuration flag, input buffers, and any other settings, and forms the sole communication channel between host and guest. Once the environment is

[22] From [68]: "The smallest unit of compute in the zkVM circuit, analogous to a clock cycle on a physical CPU. The complexity of a guest program's execution is measured in clock cycles as they directly affect the memory, proof size, and time performance of the zkVM".

ready, the host launches the *Prover*. The resulting receipt, containing both the public journal and its zero-knowledge seal, as said before, can then be forwarded to a third party, who may verify it independently without rerunning the computation or trusting the original host.

2.3 Proof System Used

The zero-knowledge layer adopted by *zkpoex* is precisely the `RISC Zero proof stack` [9]. It compresses the full execution trace discussed in RISC Zero zero-knowledge virtual machine into a single on-chain verifiable artefact, balancing three conflicting goals: transparent setup, fast GPU proving, and tiny verifier costs. The illustrative figure for this chapter and its subsections is Figure 2.3.

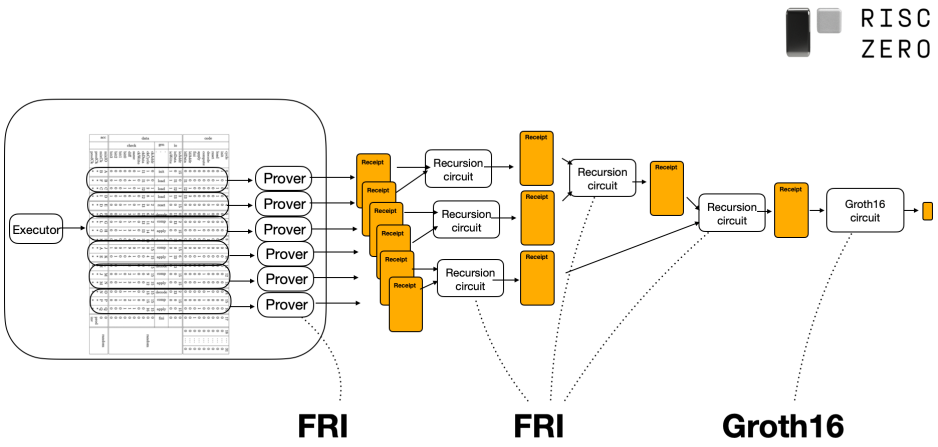

Figure 2.3: The RISC Zero's full proving stack [9]

2.3.1 Four layer architecture

Before diving into the individual stages, it is helpful to view the RISC Zero proof stack as a `pipeline`: From [71]: raw execution traces enter at the top, while a concise, on-chain–friendly SNARK emerges at the bottom. The transformation is carried out in four logical layers, each building on the previous one and serving a distinct purpose in the compression hierarchy.

1. **Execution:** The guest routine is run natively, but thanks to continuations, the VM may pause whenever memory, time, or proof limits are met, yielding a sequence of `segments`.

2. **Segment proving:** With `0STARK`, each section is independently verified; it is a STARK dialect, whose arithmetisation is AIR-based and whose commitment scheme relies on the FRI low-degree protocol over the 31-bit `Baby Bear` field. 0STARK was engineered for massive horizontal scalability on GPU clusters and currently delivers about 98 bits of concrete security.

3. **Aggregation proving:** The many segment receipts are fed into a dedicated `recursion circuit`, itself expressed in 0STARK, which folds them into a single STARK proof. Because verification is executed inside the same proof system that created the segments, recursion remains GPU-friendly, and keeps prover costs sub-quadratic in the total trace length.

4. **STARK-to-SNARK compression:** Finally, the aggregated STARK is verified inside a small `Groth16` circuit. The result is a succinct elliptic-curve SNARK, only a few hundred bytes, whose verification gas on Ethereum fits comfortably below typical block limits, making it practical for the `VerifierContract` referenced in The Smart-Contract Owner.

2.3.2 Design rationale

The layering outlined above is not arbitrary. Each component has been chosen to reconcile transparency, prover throughput, and on-chain efficiency, three properties that rarely coexist in a single proof system. The considerations below summarise why the RISC Zero team settled on this particular combination of STARKs, recursion, and Groth16.

Transparent trust model: All layers, except the final Groth16 step, are STARK-style and therefore do not require `trusted setup`; the only structured reference string left is the universal SRS[23] provided with Groth16, which can be reused by any application.

Proof size vs. prover cost: While a pure STARK would avoid elliptic curves altogether, its proof would weigh hundreds of kilobytes. Compressing the proof through Groth16 trades a one-off trusted setup for two decisive advantages:

(i) A 1 kB receipt;

(ii) A verifier that fits into a single precompile on most EVM chains.

[23] A structured reference string (SRS) is the public parameter set produced in a one-time trusted-setup ceremony; Groth16 needs a universal SRS, whereas STARKs require none.

GPU parallelism: Running FRI over a small prime field allows the prover to exploit SIMD arithmetic, and FFT kernels are already optimised for graphics workloads. Recursion is structured so that each segment can start proving as soon as its trace is available, the executor never has to wait for the last segment to finish before dispatching earlier ones, maximising cluster utilisation.

2.3.3 Implications for *zkpoex*

Embedding *zkpoex* in this proof stack yields concrete benefits:

- **Auditability:** Anyone can verify the final Groth16 receipt on-chain and be certain that the underlying STARKs were produced from the exact *context_state* and *program_specification* hashes committed in the journal, see the hash-matching logic in The Smart-Contract Owner.
- **Privacy:** The aggregation circuit reveals only the public journal; all intermediate segment traces, including the exploiter's calldata, remain hidden from both the bounty sponsor and the chain.
- **Scalability:** Thanks to continuations and GPU-centric 0STARK, even complex EVM transactions (or entire blocks) can be proven within the time and memory budget of commodity cloud instances.

In short, the RISC Zero proof system equips *zkpoex* with a battle-tested, transparent and economically viable foundation, ensuring that every exploit proof produced by The *zkpoex* Prover can be verified efficiently and trustlessly on any EVM-compatible network.

2.4 The Rust EVM

A zero-knowledge executor that must replay arbitrary Ethereum bytecode inside a Rust-centric zkVM, like the RISC Zero, one gains a decisive advantage from an interpreter developed in the same language. The *Rust EVM* crate[24] maintained by the Rust-Ethereum working group offers a compact, no-*unsafe* implementation that meets this requirement [72]. The following pages explain why this choice was preferred over C++, Go, or JavaScript engines, describe the interpreter's essential building blocks, and clarify the adjustments applied for *zkpoex*. For the reader's convenience, the discussion refers back to the execution semantics introduced in 1.4.

[24] In Rust, a *crate* is the smallest compilation unit, a library or binary package published on `crates.io` and imported with `cargo`.

2.4.1 Why a Rust interpreter

The first reason is tool-chain homogeneity. Every layer of the RISC Zero stack, guest SDK, host interface, proof gadgets, already relies on Rust; by using a Rust interpreter,r the project avoids foreign-function interfaces, keeps the guest binary small, and reaps deterministic builds through a single `cargo` command. Memory safety follows as a corollary: ownership rules guarantee that no dangling pointer or data race can corrupt the trace, an essential property because a crash inside the zkVM would render the proof invalid. Equally important is the interpreter's modular approach to hard forks. Gas tables and pre-compile behaviour vary from Frontier to Shanghai, yet the crate encapsulates these differences behind "handler" traits so that *zkpoex* can select the appropriate ruleset at compile time with no change to opcode semantics. Finally, the code base is deliberately lightweight; apart from primitive numeric types and hashing utilities, there are no heavy external dependencies, which simplifies static linking into the guest image and minimizes proof size.

2.4.2 Interpreter design

At run time, the engine revolves around three subsystems. A tight dispatch loop maps opcodes to their semantic handlers while accounting for fork-specific gas schedules; whenever the remaining gas falls below a safety margin, the zkVM yields, allowing the host to seal the current segment before memory usage grows. State access is mediated by a host trait that abstracts balance, storage, and code look-ups. In *zkpoex*, this trait is backed by a Merkle-tree store pre-seeded from the `context_state`; every leaf that the interpreter touches is logged in the journal so the verifier can recreate the same root. Cold-versus-warm bookkeeping, introduced by `EIP-2929`, is maintained by an in-guest cache whose final digest appears in the receipt, ensuring that gas metering remains deterministic on-chain.

2.4.3 Tailoring the engine to *zkpoex*

Deterministic gas accounting is the foremost requirement: all host-dependent look-ups are performed in-guest and their outcomes are committed to the public output so that verification reproduces the exact trajectory. Opcode coverage extends to every instruction up to the Shanghai hard fork, which suffices for contracts deployed before Cancun; when newer opcodes become provable within RISC Zero, the handler table can be updated without affecting archival proofs. Interpreter failures are translated into a dedicated journal flag rather than a process abort, enabling the verifier to distinguish genuine contract reverts from internal malfunctions. Together, these adap-

tations allow the Rust EVM to reproduce, byte for byte, the semantics and logic described in 1.4 while fitting comfortably within the resource budget of the zkVM.

2.5 Exploit Execution Flow

The pathway from "crafted calldata on a laptop" to an auditable bounty claim is driven by the `evm_runner` program, which will be further explored in the Implementation and Validationrd chapter of this thesis. Internally, that runner respects a fixed address map so that the guest logic never has to be re-linked when the auditor changes contracts:

- **0x7A46E7...00**: the *Target*. Whatever contract is under scrutiny is deployed at this address.
- **0xCA11E4...00**: the *Caller*. This externally owned account originates the malicious transaction.
- **0xE4C2...00**: a pre-loaded `ContextTemplateERC20` used by several reference exploits.
- **0x1000...aa - 0x1000...ff**: a sparse range reserved for any extra helper contracts the prover may wish to preload.

With those conventions in mind, the execution flow unfolds in four stages.

1. **Preparation:** The auditor spins up a private Ganache (or Anvil) fork anchored at the last safe block, deploys the Target to `0x7A46...00`, and funds the Caller account at `0xCA11E4...00`. A JSON snapshot, `context_state.json`, captures every account touched by the forthcoming call, and a second file, `program_spec.json`, records the post-conditions that *must* hold if no exploit exists. Calldata is then assembled from the relevant ABI signature; a typical example for the running tests is `16112c6c...00` calldata (the selector for `exploit(bool)`) for the *BasicVulnerable* example that is shown more in depth in section 3.4.1. All blobs are hashed and logged so that later steps can prove consistency without revealing their contents.
2. **Proving:** The host embeds the snapshot, specification, calldata, and optional value transfer into the zkVM environment and launches the RISC Zero prover. Throughout execution, the guest checks that every *fixed* predicate in the specification already holds, guaranteeing that any later violation truly stems from the transaction being proved.

3. **Receipt generation:** When the trace ends, the zkVM returns a binary receipt whose public journal carries three items:

   (a) a Boolean flag signalling that at least one predicate failed (`true` = exploit found);

   (b) Keccak256(S), the hash of the program specification actually enforced;

   (c) Keccak256(C), the hash of the supplied context state.

   No opcode log, call stack, or raw calldata leaks, only their hashes.

4. **Verification:** The CLI can stop here (offline verification) or push the receipt to an on-chain `VerifierContract`. The contract recomputes the two hashes from today's artefacts and invokes a `verify()` function. Success proves not only that the exploit drains the Target under the published state, but also that the input blobs match those posted by the owner. Because the Caller's private key resides off-chain, the concrete attack vector remains hidden while the white hacker automatically receives the reward.

Empirically, these figures show that *zkpoex* a laptop-class machine or a short Bonsai job is all that is needed to convert a confidential exploit into a succinct, publicly verifiable proof.

Chapter 3

Implementation and Validation

The third chapter details how the *zkpoex* prototype was turned from a design into a working proof-of-concept (PoC). The narrative follows the data as it travels from a local fork of `geth`, through a purpose-built Rust EVM runner, into the Risc0 zero-knowledge virtual machine and finally back on-chain inside the `VerifierContract`. Particular care is paid to the points where an ordinary execution trace is converted into a zero-knowledge receipt so that an external observer can tell, without guessing, that an exploit really exists.

3.1 Development Environment

All components are written in stable Rust 1.86.0 and organised in a multi–crate workspace stored at [11]. The manifest reveals five distinct crates, each mapped to a well-defined concern:

```
[workspace]
resolver = "2"
members  = [
  "shared",         # common utilities and domain types
  "evm-runner",     # pure-Rust EVM interpreter plus spec checker
  "guest",          # code executed inside the zkVM
  "host",           # CLI that drives proving
  "methods",        # auto-generated Risc0 bindings
]
```

Why five crates rather than a monolith?

- *shared* is `no_std`-friendly and contains only data structures, hashing helpers, and logging macros that are reused by every other crate. Isolating it keeps the guest's dependency tree minimal.

- *evm-runner* links the `evm (0.42.0)` interpreter and all predicate checking code. It targets `std` so that the same logic can be fuzz-tested natively before being embedded in the zkVM.
- *guest* is compiled for the `riscv32imac-unknown-none-elf` target; splitting it out ensures the guest ELF contains no accidental host-only dependencies and gives it a stable *Image ID*.
- *host* drives proof generation, serialises inputs, and (optionally) pushes the receipt to the on-chain `VerifierContract`. Keeping it separate allows faster incremental builds because it never needs a RISC-V cross-compile.
- *methods* is produced automatically by `risc0-build`; it exports the const `ZKPOEX_GUEST_ELF` and the matching `IMAGE_ID` is used both by the prover and by verifiers.

This breakdown shortens compile times, limits the guest's code size, and clarifies the trust boundaries: only *guest* runs in the STARK circuit, while *host* and *evm-runner* can evolve without forcing a re-audit of the zero-knowledge core.

3.1.1 Toolchain

Every machine that builds *zkpoex* uses the exact same Rust compiler and components. The file `rust-toolchain.toml` placed at the workspace root tells `rustup` which toolchain to install:

```
[toolchain]
channel    = "1.86.0"          # stable compiler known to work
profile    = "minimal"         # std, rustc, cargo only
components = [
  "rustfmt",                   # formatting gate in CI
  "clippy",                    # lints treated as errors
  "rust-src"                   # needed to cross-compile the guest
]
```

These are the reasons why this file is important:

- Guarantees that the guest ELF, and thus its Risc0 image ID, is reproduced bit-for-bit on every host.
- Packages only the pieces required for the project, keeping Docker images small and CI jobs fast.
- Adds `rust-src` so that `cargo build --target x-unknown-none-elf` works out of the box when compiling the zkVM guest.

With the toolchain pinned, a fresh setup is reduced to three commands:

```
$ rustup toolchain install 1.86.0  # download the pinned compiler
$ rustup default 1.86.0  # active the toolchain
$ cargo build --release  # builds all the crates
$ cargo test             # runs native EVM tests
```

No nightly features, external linkers, or patched forks are required with this configuration.

3.1.2 Key External Libraries

Version 0.1.0 of *zkpoex* (April 2025) relies on only three upstream crates to generate and validate a proof. Pinning them at the exact versions shown below ensures that every audit can rebuild the binary bit-for-bit and that later ecosystem upgrades cannot silently alter the circuit.

Crate	Version	Role in *zkpoex* v0.1.0
`evm`[1]	0.42.0	Canonical byte-for-byte interpreter embedded in the *evm-runner* crate; replays the transaction locally exactly as it would execute on mainnet before the trace enters the zkVM.
`risc0_zkvm`[2]	2.0.0	Supplies the 0STARK prover, guest SDK and Groth16 compression pipeline; every receipt produced by *zkpoex* is ultimately a Risc0 proof.
`primitive-types`	0.12.0	Single source of fixed-width Ethereum primitives (`U256`, `H160`, `H256`); guarantees identical encodings on both host and guest.

Table 3.1: Pinned library set for the first public release.

Why no other crates appear here Utilities such as `serde`, `tokio`, or `clap` are secondary crates. They make the CLI convenient but carry no cryptographic weight. Should one of them receive a breaking change, it affects only peripheral code; the core proof remains verifiable so long as the three crates above stay frozen at the versions listed in Table 3.1.

[1] Built with the `with-serde` feature so that full `MemoryAccount` objects can be serialised across the host-guest boundary.

[2] Uses the FRI backend shipped by Risc0; the resulting STARK is compressed to Groth16 for low-gas on-chain verification.

3.1.3 Auxiliary tooling

Foundry is used only to compile the Solidity targets that form the Case Study suite; bytecode and ABI files are committed under the `bytecode/` folder so that every reviewer can rebuild the PoC without installing a global Solidity toolchain.

The day-to-day interface, however, is the `justfile` shipped in the repository root. While it will be deprecated in the next releases, the current script distils every common workflow compiling contracts, spinning up a local fork, generating a proof, pushing it on-chain into a single shell-friendly command.

- **Unit testing** → `$ just test-evm`
  It compiles the contracts and runs every `#[test]` declared in the *evm-runner* crate, proving that each case study really violates its specification at the pure EVM level before any zero-knowledge overhead is added.

- **Local proving** → `$ just prove`
  wraps the entire *host* call in a single command; every argument maps one-to-one to a flag in `host/main.rs`:

  - `function:` canonical Solidity selector, e.g. `\withdraw(uint256)"`.
  - `params:` comma-separated ABI values fed to the selector.
  - `context_state:` path to the JSON file that fixes all accounts and storage slots present *before* the call.
  - `program_spec:` path to the JSON array of post-conditions the guest must try to violate.
  - `value:` ETH (in wei) sent along with the call if needed; use `0` for pure logic bugs if it is not needed.
  - `network` `local`, `testnet` (Holesky) or `mainnet`; selects the RPC endpoint and the canonical Risc0 verifier address.
  - `bonsai` (optional, default `false`) set to `true` to outsource proving to Bonsai instead of the local GPU.
  - `verbose` (optional, default `false`) enables full `tracing` output from both host and guest.

- **Bonsai proving:** As told before, adding `bonsai=true` switches the prover to Risc0's cloud service. The script sets `RISC0_DEV_MODE=0` automatically and exports the API key from `BONSAI_API_KEY`.

- **On-chain verification:** → `$ just onchain-verify`
  run the file `onchain_verifier.rs` from *host* crate and reads the

`receipt.bin` saved before in the file system after a proof generation, encodes the seal and journal, and submits the Smart-Contract `verify()` call to the chosen network.

Here an example of a single-shot invocation that proves the `exploit(bool)` bug in the *BasicVulnerable* contract against a local `anvil` fork:

```
$ just prove "exploit(bool)" true \
    ./shared/examples/basic-vulnerable/context_state.json \
    ./shared/examples/basic-vulnerable/program_spec.json \
    0 local bonsai=false verbose=true
```

Please note that within the justfile, it is possible to run the 3 built-in examples with a more streamlined and readable command → `$ just example-name-of-example-prove`. In Figure 3.1, an example of execution from the CLI.

Newcomers need only skim `justfile` to discover the available actions. The file keeps all environment variables in one place, validates mandatory secrets (`WALLET_PRIV_KEY`, `BONSAI_API_KEY`) before running, and prints a concise banner with all the logs related to internal script executions to let the user know how the software is running at that time. The *justfile* makes the PoC reproducible with nothing more than `cargo`, `solc`, and `just`.

3.2 EVM-side Development

Implementing an Ethereum interpreter inside a zero-knowledge circuit forces every allocation, hash, and gas charge to be explicit. The crate `evm-runner` therefore wraps the canonical `evm` crate (v 0.42.0) in a thin adapter that:

(i) reconstructs states from JSON;

(ii) enforces a fixed address schema;

(iii) evaluates a purpose-built predicate engine.

The following subsections dissect that adapter.

3.2.1 Fixed Address Schema

All contracts are deployed at compile-time constants so that the guest binary never changes when the auditor swaps artefacts. Below is the fixed address scheme (which we have already mentioned in Section 2.5 of the previous chapter) to provide a somewhat more rigorous generalization for simulating the exploit within the EVM. Please note that it is not necessary to use the same addresses as the smart contracts we want to examine or verify

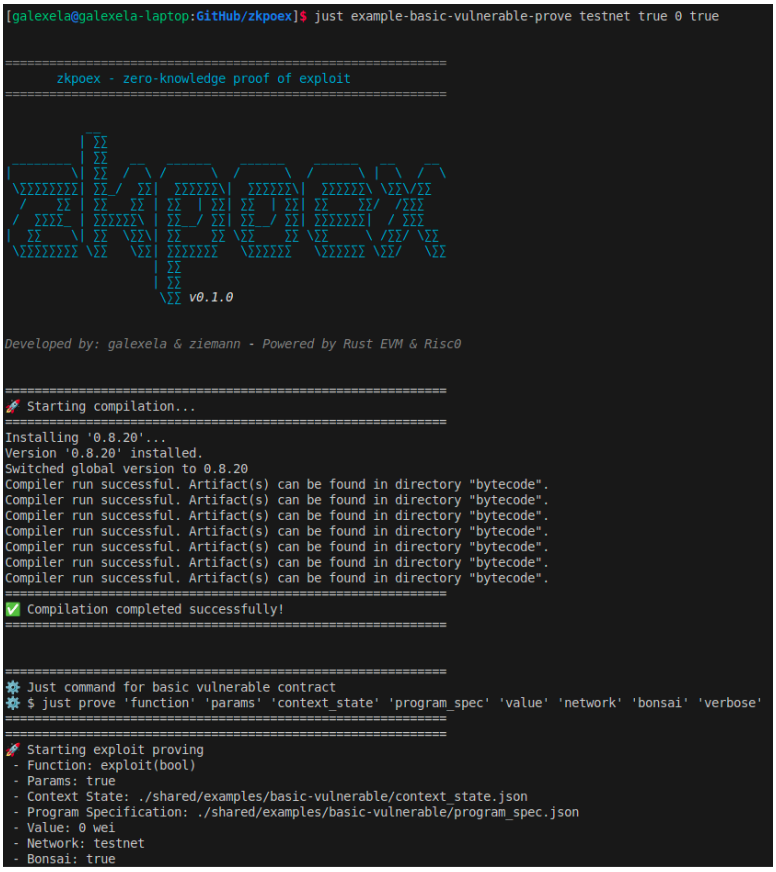

Figure 3.1: Terminal output for the `just prove` recipe. The script compiles the guest, executes the EVM trace, generates a STARK proof, and finally emits a 304-byte Groth16 receipt ready for on-chain verification.

on-chain, since this verification is done on the bytecodes corresponding to the smart contracts. These addresses have been standardized for readability and simplicity. More accurate standardization for more complex case studies will certainly be done and improved over the current one in later versions of *zkpoex*.

```
const TARGET_ADDRESS  : &str =
            "7A46E7000000000000000000000000000000000000";
const CALLER_ADDRESS  : &str =
            "CA11E4000000000000000000000000000000000000";
const TEMPLATE_ADDRESS: &str =
            "E4C2000000000000000000000000000000000000";
const RESERVED_ARBITRARY_ADDRESSES: (&str,&str) =
    ("1000000000000000000000000000000000000000",
     "100000000000000000000000000000000000fff");
```

```
10
11                                                    evm-runner/lib.rs
```

The two hard-coded actors mirror a normal exploit: an externally owned account (`CALLER_ADDRESS`) initiates a call to the vulnerable contract (`TARGET_ADDRESS`). A contiguous helper range is left free for auxiliary byte-code, such as the `AttackContract` of the reentrancy case study (see 3.4.3). Because the linker can resolve every symbol at build time, the guest ELF has a stable image ID, and the on-chain verifier needs to remember only a single hash.

3.2.2 State Reconstruction with `MemoryAccount`

The host serialises each account into a plain JSON object (the *context_state*). Inside the guest, these objects are turned back into the type `MemoryAccount` provided by the `evm` crate [73]. The function `build_global_state` performs the conversion.

```
1  pub fn build_global_state(
2      map: &mut BTreeMap<H160, MemoryAccount>,
3      accounts: &Vec<AccountData>)
4  {
5      for a in accounts {
6          map.insert(
7              H160::from_str(&a.address).unwrap(),
8              MemoryAccount {
9                  nonce:   a.nonce,
10                 balance: a.balance,
11                 storage: a.storage.clone(),
12                 code:    a.code.clone(),
13             });
14     }
15 }
16                                                    evm-runner/lib.rs
```

Every entry contained in `context_state` is parsed into an in-memory `MemoryAccount` [73]. That structure exposes the same four fields described in the Ethereum Yellow Paper: *nonce*, *balance*, *storageRoot* and *codeHash*, with the storage trie materialised as a flat hash-map from 256-bit keys to 256-bit words, and the code hash backed by the raw byte-code itself.

From the resulting vector, the loader builds *two* completely independent maps of type `BTreeMap<H160, MemoryAccount>`:

- `pre_state` is passed to the `StackExecutor`. Every state-changing opcode (`SSTORE`, `CALL`, gas accounting) mutates this map, so by the end of `transact_call` it contains the *post-execution* world.

- `pre_state_snapshot` is a deep clone taken immediately after `pre_state` is built. No runtime code ever touches it, guaranteeing that it remains an exact byte-for-byte record of the *pre-execution* state.

After the call returns, the program can now execute a transaction on the cloned state that we created.

3.2.3 Transaction simulation using `transact_call()`

The entry point `run_evm` hands execution to the Stack-based interpreter exported by the `evm` crate, selects the Cancun gas schedule via `Config::cancun`. It wires the reconstructed state into a `StackExecutor` [74] and calls `transact_call()`. It accepts an `ExitReason`; only `STOP` or `RETURN` are accepted. If the output is `STOP`, it means that the machine encountered an explicit stop; if it is *RETURN*, the machine encountered an explicit return. Reverts abort the proof so that an attacker cannot skip predicate checks by exhausting gas. The core invocation is:

```
let (exit, _) = executor.transact_call(
        initiator,              // the external caller
        recipient,              // the vulnerable contract
        value,                  // ETH amount sent with the call
        hex::decode(calldata).unwrap(),// raw ABI-encoded calldata
        u64::MAX,               // upper gas bound
        Vec::new());            // no access-list (is empty)

assert!(matches!(exit,
        ExitReason::Succeed(ExitSucceed::Stopped) |
        ExitReason::Succeed(ExitSucceed::Returned)));

                                                    evm-runner/lib.rs
```

The helper builds a *message call* object that contains:

- the sender and destination addresses;
- the transferred `value`;
- an input holding the ABI-encoded function selector and arguments (the calldata);
- a gas stipend, here set to $2^{64} - 1$ so the interpreter itself performs metering.

After this function call, we will get the *post_state* from the executor, which we will use to compare with the *pre_state* as shown in subsection 3.2.5.

3.2.4 Predicate Engine and Condition Language

The term *predicate engine* refers to the block of functions that, once the EVM finishes, decides whether the observed state transition satisfies the rules laid down by the smart-contract owner. Those rules live in a JSON file called `program_spec`, introduced in the Program Specifications section. After parsing, the file becomes a vector of `MethodSpec` values, one per function selector. Each value carries a list of predicates that must hold *after* the selector has been executed.

Four predicate flavours cover the patterns most frequently needed during audits:

- *Fixed.* Compare a single post-state field-balance, nonce, or storage slot to a constant.
- *Relative.* Compare two fields, one pulled from the snapshot (s) and one from the live post-state (s'); an optional arithmetic operator can be applied to s' first.
- *Input-dependent fixed.* Like the fixed case, but the constant is supplied at runtime through calldata, useful for "must refund at least *amount*" assertions.
- *Input-dependent relative.* Combine the previous two ideas: compare a pre-state field to a function of (s', calldata).

Whatever the flavour, evaluation eventually funnels down to the generic helper shown in Listing 3.7. The two operands arrive already converted to the same concrete type, so the comparator can rely on the `PartialOrd` trait alone. That abstraction lets the engine handle 256-bit integers, ordinary `u64` counters, or booleans with one shared implementation.

```
fn check_condition_op<T: PartialOrd + std::fmt::Debug>(
    operator: &Operator,
    first_val: T,
    second_val: T,
) -> bool {
    shared::log_debug!(
        "Checking condition {:?} {:?} {:?}",
        operator,
        first_val,
        second_val
    );
    match operator {
        Operator::Eq => first_val == second_val,
        Operator::Neq => first_val != second_val,
        Operator::Gt => first_val > second_val,
```

```
16          Operator::Ge => first_val >= second_val,
17          Operator::Lt => first_val < second_val,
18          Operator::Le => first_val <= second_val,
19      }
20  }
21
22                                                  evm-runner/lib.rs
```

When the engine reaches the end of the list, it returns a Boolean flag called `exploit_found`. A single failing predicate flips the flag to `true`; the remaining checks are skipped to save cycles. The flag, together with the *Keccak256* hashes of the loaded specification and context state, forms the journal that the guest commits to the Risc0 receipt. The verifier contract, therefore, needs no knowledge of individual predicates: a mismatch in either hash is enough to reject stale or tampered inputs, while a *true* flag certifies that at least one owner-defined safety property was violated in the simulated transaction.

3.2.5 State validation and post-execution proof

Once the predicate language (Section 3.2.4) has been parsed, the runner performs two consecutive passes over the data: a *pre-flight* sanity check that guarantees the experiment starts from a coherent state, and a *post-state* inspection that decides whether the simulated transaction violates at least one owner-supplied invariant.

Pre-flight scan

The helper `verify_pre_state` walks every `MethodSpec` contained in the parsed `program_spec`. Only *fixed* predicates are evaluated here because they speak exclusively about the initial state s. At the same time, the routine checks two structural properties of the sandbox:

- exactly one account bears the reserved `CALLER_ADDRESS` and one bears `TARGET_ADDRESS`;
- if an account entry embeds bytecode, the same bytecode is already loaded in the live `MemoryBackend`, preventing the prover from crafting exploits against phantom contracts.

```
1  fn verify_pre_state(state: &MemoryStackState<MemoryBackend>,
2                      spec:  &Vec<MethodSpec>) {
3      for m in spec {
4          for cond in &m.conditions {
5              if let Condition::Fixed(f) = cond {
```

```
                assert!(check_fixed_condition(state, f),
                        "pre-state violates {:?}", f);
            }
        }
    }
}
```

evm-runner/lib.rs

A failure at this stage aborts the run before any gas is spent: the guest cannot proceed to proving until the owner's baseline requirements are met.

Post-state inspection

After the EVM step completes, the prover holds two concrete world images: the immutable snapshot taken just before the call and the mutable map that now captures the aftermath. `prove_final_state()` (Listing 3.9) receives both maps together with the entire `program_spec`. It first filters the specification by the four-byte selector so that only predicates relevant to the executed function are evaluated. The loop stops at the first violation, at which point `exit_succeed` flips to `true`; the caller treats that bit as "Exploit found".

```
fn prove_final_state(
    pre_state: &MemoryStackState<MemoryBackend>,
    post_state:&MemoryStackState<MemoryBackend>,
    program_spec:&Vec<MethodSpec>,
    calldata:   &str,
) -> bool {
    let mut exit_succeed = false;
    let (method_conditions, method_arguments) =
        filter_program_spec(program_spec, &calldata[..8]);

    for condition in method_conditions {
        match condition {
            Condition::Fixed(condition) => {
                let result =
                check_fixed_condition(post_state, &condition);
                if !result {
                    shared::log_info!("Fixed condition
                    failed: {:?}", condition);
                    exit_succeed = true; break;
                }
            }
            Condition::Relative(condition) => {
                let result =
                check_relative_condition(pre_state,
```

```
                post_state, &condition);
                if !result {
                    shared::log_info!("Relative condition
                    failed: {:?}", condition);
                    exit_succeed = true; break;
                }
            }
            Condition::InputDependantFixedCondition(condition) => {
                let result = check_input_dependant_fixed_condition(
                    post_state, &condition, calldata,
                    method_arguments.clone());
                if !result {
                    shared::log_info!("Input-dependant fixed
                    condition failed: {:?}", condition);
                    exit_succeed = true; break;
                }
            }
            Condition::
            InputDependantRelativeCondition(condition) => {
                let result =
                check_input_dependant_relative_condition(
                    post_state, &condition, calldata,
                    method_arguments.clone());
                if !result {
                    shared::log_info!("Input-dependant relative
                    condition failed: {:?}", condition);
                    exit_succeed = true; break;
                }
            }
        }
    }
    exit_succeed        // true -> at least one predicate failed
}

                                                    evm-runner/lib.rs
```

The individual predicate checkers are thin adapters that fetch the required values and delegate comparison to the shared operator table `check_condition_op()`. All code is reproduced verbatim from *evm-runner/lib.rs*.

1. *Fixed condition:* The helper fetches one field from the *post-state* and compares it to an immutable constant carried in the JSON specification.

```
fn check_fixed_condition(
    state: &MemoryStackState<MemoryBackend>,
    fixed_condition: &FixedCondition,
) -> bool {
```

```
    let state_key =
    fixed_condition.k_s.split('.').collect::<Vec<&str>>();
    let value = get_state_value(state, state_key);
    check_condition_op(&fixed_condition.op,
    value, fixed_condition.v)
}


                                                evm-runner/lib.rs
```

2. *Relative condition:* Two fields are read, one from the immutable snapshot and one from the live map. An optional arithmetic operator adjusts the right-hand term before the comparison.

```
fn check_relative_condition(
    pre_state:  &MemoryStackState<MemoryBackend>,
    post_state: &MemoryStackState<MemoryBackend>,
    relative_condition: &RelativeCondition,
) -> bool {
    let pre_state_key  =
    relative_condition.k_s.split('.').collect::<Vec<&str>>();
    let post_state_key =
    relative_condition.k_s_prime.split('.')
    .collect::<Vec<&str>>();

    let pre_state_value  =
    get_state_value(pre_state,  pre_state_key);
    let post_state_value =
    get_state_value(post_state, post_state_key);

    let second_val = match &relative_condition.value_op {
        Some(op) => {
            let v = relative_condition.v
                        .expect("v must accompany value_op");
            execute_condition_op(op, post_state_value, v)
        }
        None => post_state_value,
    };

    check_condition_op(&relative_condition.op,
    pre_state_value, second_val)
}


                                                evm-runner/lib.rs
```

3. *Input-dependent fixed condition:* Here, the constant is not hard-coded: it is extracted from calldata at run-time, allowing the same rule to serve multiple parameter sets. The present implementation covers only plain

`uint256` arguments and is meant as a stepping-stone toward richer encodings.

```
fn check_input_dependant_fixed_condition(
    state: &MemoryStackState<MemoryBackend>,
    input_dependant_fixed_condition:
    &InputDependantFixedCondition,
    calldata: &str,
    method_arguments: Vec<MethodArgument>,
) -> bool {
    let state_key = input_dependant_fixed_condition.
    k_s.split('.').collect::<Vec<&str>>();
    let value = get_state_value(state, state_key);
    let input_value = get_input_value(
        calldata,
        method_arguments,
        input_dependant_fixed_condition.input.clone(),
    );
    check_condition_op(&input_dependant_fixed_condition.op,
    value, input_value)
}

                                                evm-runner/lib.rs
```

4. *Input-dependent relative condition:* This variant generalises the previous one: it derives the comparison term from both the mutated storage and the caller-supplied parameter, then applies an arithmetic operator before the final check. Like the fixed counterpart, support is limited to simple integer types and will be extended as more complex attack surfaces are explored.

```
fn check_input_dependant_relative_condition(
    state: &MemoryStackState<MemoryBackend>,
    input_dependant_relative_condition:
    &InputDependantRelativeCondition,
    calldata: &str,
    method_arguments: Vec<MethodArgument>,
) -> bool {
    let pre_state_key  =
    input_dependant_relative_condition.k_s.split('.')
    .collect::<Vec<&str>>();
    let post_state_key =
    input_dependant_relative_condition.k_s_prime.split('.')
    .collect::<Vec<&str>>();

    let pre_state_value  =
    get_state_value(state, pre_state_key);
    let post_state_value =
```

```
18      get_state_value(state, post_state_key);
19
20      let input_value = get_input_value(
21          calldata,
22          method_arguments,
23          input_dependant_relative_condition.input.clone(),
24      );
25      let second_val = execute_condition_op(
26          &input_dependant_relative_condition.input_op,
27          post_state_value,
28          input_value,
29      );
30
31      check_condition_op(
32          &input_dependant_relative_condition.op,
33          pre_state_value,
34          second_val,
35      )
36  }
37
38                                                  evm-runner/lib.rs
```

Each helper finishes by delegating to `check_condition_op()` (Listings 3.7), which implements the six relational operators with uniform logging. The two input-dependent variants are intentionally minimal; they work for current proofs yet leave room for future support of tuples, dynamic arrays, and packed structs once more sophisticated exploits need to be modelled. The only case study that uses an input-dependent variant is *reentrancy attack* (in Section 3.4.3).

3.3 RISC Zero zkVM-side Development

This section traces the data once it leaves the Rust EVM runner and enters the RISC Zero proving stack. First, the host CLI serialises the execution context and launches the prover. Next, the guest program replays the transaction inside the zkVM and commits a compact journal. Finally, the host either stores the receipt locally or forwards it to the on-chain `VerifierContract`. Each phase is illustrated with the code drawn from `host/main.rs` and `guest/main.rs`.

3.3.1 Serialising the proving payload

The CLI aggregates every artifact, calldata, JSON snapshots, blockchain settings, and the optional `value` into one struct that the guest can deserialize in a `no_std` context. In Listings 3.14, the input payload written to the guest

```
#[derive(Serialize, Debug)]
struct InputData<'a> {
    calldata: &'a str,
    context_state: String,
    program_spec: String,
    blockchain_settings: String,
    value: U256,
}
                                                    host/main.rs
```

A single call to the `ExecutorEnv` builder copies this struct into guest memory, guaranteeing that every byte influencing the proof travels through one well-defined channel.

```
let env = ExecutorEnv::builder()
        .write(&input)
        .unwrap()
        .build()
        .unwrap();
                                                    host/main.rs
```

3.3.2 Launching the prover

After the `InputData` struct has been serialised into guest memory, the host invokes the prover. The zkVM first replays the guest ELF, records a full STARK trace, and then, because we pass `ProverOpts::groth16()`, compresses that trace into a one-kilobyte Groth16 seal. Alongside the seal, the `prove_with_ctx` call returns per-run statistics such as total cycle count, memory peak, and wall-clock duration, which are printed if the CLI is launched in verbose mode.

```
let prove_info = default_prover().prove_with_ctx(
        env,
        &risc0_zkvm::VerifierContext::default(),
        ZKPOEX_GUEST_ELF,
        &risc0_zkvm::ProverOpts::groth16(),
).unwrap();
let receipt = prove_info.receipt;
                                                    host/main.rs
```

3.3.3 Guest execution and journal construction

Inside the zkVM, the guest deserialises the three JSON blobs and the attached `value`, invokes `run_evm`, then commits three public fields to the journal.

```
    let result = run_evm(
        &calldata,
        context_state,
        program_spec,
        &blockchain_settings,
        value,
    );

    let exploit_found: bool = result[0] == "true";
    let program_spec_hash: B256 =
        B256::from_str(&result[1])
        .expect("Invalid hex for program_spec_hash");
    let context_state_hash: B256 =
        B256::from_str(&result[2])
        .expect("Invalid hex for context_state_hash");

    let input = PublicInput {
        exploitFound: exploit_found,
        programSpecHash: program_spec_hash,
        contextStateHash: context_state_hash,
    };

    let encoded = PublicInput::abi_encode(&input);
    env::commit_slice(&encoded);

                                                    guest/main.rs
```

The guest publishes only three ABI-encoded fields:

- `exploitFound`: a single Boolean that becomes a 32-byte word when padded by the Solidity ABI;
- `programSpecHash`: a 32-byte Keccak-256 digest of the specification used during proving;
- `contextStateHash`: a 32-byte Keccak-256 digest of the reconstructed account snapshot.

Because each item occupies exactly one 32-byte slot, the entire public journal is $3 \times 32 = 96$ bytes. No calldata, opcode trace, storage diff, or gas meter reading leaves the circuit. Every internal detail remains hidden inside the STARK witness.

3.3.4 Extracting seal and journal

When the zkVM returns a receipt, the host must turn its two-byte arrays: the Groth16 seal and the 96-byte journal-into calldata that matches the

ABI of `VerifierContract.verify(address, bytes, bytes)`. Only the seal needs modification: the function in Listing 3.18 prefixes the raw proof with the 4-byte selector retrieved from the verifier parameters produced by the RISC Zero build, leaving the journal untouched. This wrapper lets auditors submit the transaction with a standard wallet and guarantees that Solidity receives exactly the byte layout the compiler expects.

```
pub fn encode_seal(receipt: &risc0_zkvm::Receipt)
       -> Result<Vec<u8>, anyhow::Error> {
    let seal = match receipt.inner.clone() {
        InnerReceipt::Groth16(r) => {
            let selector = &r.verifier_parameters.as_bytes()[..4];
            let mut out  = Vec::with_capacity(selector.len()
            + r.seal.len());
            out.extend_from_slice(selector);
            out.extend_from_slice(r.seal.as_ref());
            out
        }
        InnerReceipt::Fake(r) => {
            let selector = &[0xFFu8; 4];
            let mut out  = Vec::with_capacity(selector.len() + 32);
            out.extend_from_slice(selector);
            out.extend_from_slice(r.claim.digest().as_bytes());
            out
        }
        _ => bail!("Unsupported receipt type"),
    };
    Ok(seal)
}

                                                    shared/lib.rs
```

With seal and journal ready, the host signs a call to the verifier as shown in Listing 3.19; the asynchronous helper pushes the payload to the chosen RPC endpoint and prints a block-explorer link.

```
let onchain_journal = receipt.journal.bytes.clone();
let onchain_seal = encode_seal(&receipt)?;
onchain_verifier::verify_onchain(
        &private_key_str,
        &eth_rpc_url_str,
        &contract_address_str,
        onchain_seal,
        onchain_journal,
).await?;

                                                    host/main.rs
```

On-chain, the contract in Listing 3.20 first asks the low-level RISC Zero verifier to validate the seal against the agreed `imageId` and the SHA-256 digest of the journal. It then decodes the journal, checks that `exploitFound` is `true`, and compares the two Keccak hashes with the values stored during deployment. A triple match triggers the `ExploitFound` event and transfers the bounty; any mismatch reverts.

```
function verify(address beneficiary, bytes calldata seal,
                bytes calldata journal) public payable {
    risc0_verifier_contract.verify(seal, imageId, sha256(journal));
    (bool exploitFound, bytes32 specHash, bytes32 ctxHash) =
    abi.decode(journal,(bool,bytes32,bytes32));
    require(exploitFound, "exploit not found");
    require(specHash == program_spec_hash, "wrong spec");
    require(ctxHash == context_state_hash, "wrong context");
    emit ExploitFound(beneficiary, address(this));
    require(payable(beneficiary)
    .send(REWARD_IN_ETH), "payment failed");
}

                                    contracts/VerifierContract.sol
```

The entire public payload is ninety-six bytes, one padded Boolean plus two Keccak hashes, so no sensitive trace information ever leaves the circuit, yet the verifier obtains a complete, on-chain guarantee that a real exploit exists and that it was proved against the current specification and state.

3.4 Case Studies

Three contracts were ported to the fixed address space and proved inside the zkVM. Each example demonstrates a different bug class.

3.4.1 BasicVulnerable Logic

The contract below exposes two deliberately insecure entry points that allow an attacker to burn all assets, native ether, and ERC-20 tokens held by the contract itself. The bug is not subtle; nonetheless, it demonstrates how a single *fixed* predicate can certify such a flaw without revealing the calldata used to trigger it.

```
contract BasicVulnerable {
    function exploit(bool _exploit) public {
        if (_exploit)
            payable(address(0)).transfer(address(this).balance);
    }
    function exploit_erc20(bool _exploit) public {
```

```
        if (_exploit) {
            IERC20 t = IERC20(0xE4C2...);
            t.transfer(0x0000000000000000000000000000000000000001,
                       t.balanceOf(address(this)));
        }
    }
    receive() external payable {}
}

                                        contracts/BasicVulnerable.sol
```

For the method selector `16112c6c` (`exploit(bool)`) the owner publishes a single fixed predicate that forces the post-state ether balance to remain strictly positive:

```
{
  "Fixed": {
    "k_s": "7A46E7000000000000000000000000000000000.balance",
    "op": "Gt",
    "v": "0"
  }
}

                    shared/examples/basic-vulnerable/program_spec.json
```

A symmetric rule guards the ERC-20 pathway. The selector calculated: `d92dbd19` (`exploit_erc20(bool)`) points into the ERC-20 template at `0xE4C2...000`. The predicate below states that storage slot `a070...e423`, which holds the token balance of the vulnerable contract, must be strictly greater than zero after the call:

```
{
  "Fixed": {
    "k_s": "E4C2000000000000000000000000000000000000.storage
.a070964ccc7e37c9aaee36054a704be4b9644cc02d84a3a67917c0baa3cee423",
    "op": "Gt",
    "v": "0"
  }
}
                    shared/examples/basic-vulnerable/program_spec.json
```

Triggering `exploit(true)` transfers all ether to the zero address, so the balance becomes zero and violates the first predicate. Likewise, calling `exploit_erc20(true)` drains the entire token balance to `address(1)`, setting the storage slot to zero and breaking the second predicate. In both cases, the guest sets `exploitFound = true`; the hash of the specification commits the exact JSON shown above, and the verifier contract pays the bounty while the underlying calldata remains hidden.

3.4.2 Unchecked Arithmetic

Unchecked arithmetic is a classic vulnerability that vanished from ordinary Solidity code when compiler version 0.8 introduced automatic over-flow checks [75]. By adding the `unchecked` keyword, however, a developer can still re-enable wrapping semantics, usually for gas optimisation, thereby re-opening the door to unintended under-flows and over-flows.

```
contract OverUnderFlowVulnerable {
    uint256 public balance = 1000;
    function withdraw(uint256 amount) public {
        unchecked { balance -= amount; }  // under-flow to 2^256-1
    }
}

                                contracts/OverUnderFlowVulnerable.sol
```

For the selector `2e1a7d4d` (`withdraw(uint256)`) the specification contains a single fixed predicate that forbids the sentinel wrap-around value:

```
{
  "Fixed": {
    "k_s": "7A46E700...000.storage.000...000",
    "op": "Neq",
    "v": "0xffff...ffff"
  }
}

                    shared/examples/over-under-flow/program_spec.json
```

Slot 0 maps to the `balance` variable; the condition demands that the slot must *not* equal $2^{256} - 1$ after the call. Invoking `withdraw(1001)` subtracts 1001 from 1000, the subtraction wraps, the slot is set to the forbidden value, and the predicate fails. The guest therefore flags `exploitFound = true` and the proof is accepted by the verifier.

3.4.3 Reentrancy Drain

To perform a reentrancy attack, the adversary needs an auxiliary contract that can hold ether, invoke the vulnerable function, and crucially, receive control back during the external call. The `AttackContract` in Listing 3.27 fulfils this role: it deposits a chosen `amount` of ether, triggers `withdrawETH()`, and implements a `receive()` fallback that re-enters as long as the target still holds at least one ether. Because the internal balance is reset only *after* the external call, each re-entry observes the same non-zero mapping and drains another round of funds before the first context returns.

```
function withdrawETH() public {
    uint256 bal = balances[msg.sender];
    (bool ok,) = msg.sender.call{value: bal}("");
    require(ok, "fail");
    balances[msg.sender] = 0; // state updated after external call
}
                                                ReentrancyVulnerable.sol
```

```
receive() external payable {
    if (address(vuln).balance >= 1 ether) vuln.withdrawETH();
    else payable(owner).transfer(address(this).balance);
}


                                                AttackContract.sol
```

The exploit re-enters `withdrawETH` while the target's internal balance mapping still shows a positive value, allowing repeated withdrawals in a single transaction.

The selector `64dd891a` (`attack(uint256)`) has one *input-dependent relative* predicate:

```
{
  "InputDependantRelativeCondition": {
    "k_s":       "7A46E7...00.balance",
    "op":        "Eq",
    "k_s_prime": "7A46E7...00.balance",
    "input_op":  "Add",
    "input":     "amount"
  }
}

                        shared/examples/reentrancy/program_spec.json
```

The rule says "the pre-state balance of the Target must equal the post-state balance plus the parameter `amount`". Under normal, non-re-entrant execution, the contract pays out exactly the requested sum, so the equality holds. During the attack, the fallback enters `withdrawETH` again before the balance mapping is zeroed, draining more than `amount`; therefore

$$\texttt{balance}_{\text{pre}} \neq \texttt{balance}_{\text{post}} + \texttt{amount}$$

and the predicate fails. The guest sets `exploitFound = true`.

3.4.4 Implementation Insights

Practical testing on multiple bug classes surfaced several engineering patterns that simplify maintenance and improve performance. The most notable

takeaways are summarised below.

- A fixed address map removes any need for dynamic linking of real smart-contract addresses inside the guest; the resulting ELF keeps the same image ID across audits, simplifying on-chain whitelisting.
- The very same `evm` interpreter binary is compiled three times, native for unit tests, `no_std` for the zkVM guest, and fuzz-hardened for off-chain analysis, so every opcode is costed by an identical gas table in every context. Consequently, a predicate that depends on gas-sensitive state (for example, the warm-storage discount introduced by EIP-2929) will either pass or fail consistently; the prover cannot manufacture a trace that satisfies the check locally yet fails when the verifier replays the journal.
- Compressing the STARK trace with Groth16 cuts the receipt from roughly 420kB to about one kilobyte, while proof time remains unchanged because the computationally heavy lifting still occurs in the 0STARK layer.

These observations confirm that *zkpoex* can prove both trivial logic flaws and complex stateful bugs while publishing only constant-size data on-chain.

3.5 On-chain verification path

The final hop in the *zkpoex* pipeline happens entirely on–chain. After the host has generated a Groth16 receipt and wrapped it as shown in Listing 3.18, it submits one transaction that calls `VerifierContract.verify()`. The contract (Listing 3.29) is purpose-built and short enough to audit line-by-line:

1. It delegates the heavy cryptographic work to the canonical `IRiscZero Verifier`, passing the imageID fixed at deployment time and the **Keccak256** digest of the 96-byte public journal.
2. It ABI-decodes that journal into three fields and checks:
   - if the `exploitFound` is `true`;
   - if the two Keccak hashes match the policy currently stored in `program_spec_hash` and `context_state_hash`; Either mismatch causes an immediate `revert`.
3. Upon success, it emits the `ExploitFound` event and transfers a fixed bounty of 1000 wei to the beneficiary, then toggles `exploit_proof_`

`verification_locked` so that the owner can inspect and, if needed, update the target hashes before the next round.

```
contract VerifierContract is Ownable {
    //**Here main state vars**//
    constructor(
        address _risc0_verifier_contract,
        bytes32 _program_spec_hash,
        bytes32 _context_state_hash,
        address _image_id_contract
    ) Ownable(msg.sender) {
        risc0_verifier_contract =
    IRiscZeroVerifier(_risc0_verifier_contract);
        program_spec_hash = _program_spec_hash;
        context_state_hash = _context_state_hash;
        imageId = IImageID(_image_id_contract).ZKPOEX_GUEST_ID();
    }

    //**Here other code**//

    /// @notice Verifies the public input and
    /// proof (seal) from the prover.
    ///It calls the external risc0 verifier,
    ///then decodes and checks the public input.
    ///If all checks pass, it emits an ExploitFound
    ///event and transfers the reward.
    function verify(
        address beneficiary,
        bytes calldata seal,
        bytes calldata journal
    ) public payable {
        // Check if contract is locked for
        //exploit proof verification.
        require(
            !exploit_proof_verification_locked,
            "Contract is locked for exploit proof verification"
        );

        risc0_verifier_contract.verify(seal, imageId,
        sha256(journal));

        (
            bool exploit_found,
            bytes32 claimed_program_spec_hash,
            bytes32 claimed_context_state_hash
        ) = abi.decode(journal, (bool, bytes32, bytes32));

        // Check that an exploit was indeed found.
```

```
46          require(exploit_found, "Exploit not found");
47
48          // Validate that the provided hashes
49          //(after keccak256) match the stored values.
50          // or equivalently, that the context state
51          //and program spec used to generate the proof
52          // are the same as the ones stored in the contract.
53          require(
54              claimed_program_spec_hash == program_spec_hash,
55              "Invalid program spec hash"
56          );
57          require(
58              claimed_context_state_hash == context_state_hash,
59              "Invalid context state hash"
60          );
61
62          // Emit event
63          emit ExploitFound(beneficiary, address(this));
64          require(payable(beneficiary).send(REWARD_IN_ETH),
65          "Transfer failed");
66
67          // Lock the contract to prevent further
68          // exploit proof verification until it is unlocked.
69          exploit_proof_verification_locked = true;
70      }
71  }
```

An example contract deploy can be seen at `0x66fb6f...`. An example of exploit proof verification transaction can be seen at `0x71c...ce7`.

This tx calls the verifier's selector `0x5ade6633`, supplying a public journal and the Groth16 seal produced. Etherscan shows that:

1. the low-level Risc0 precompile accepts the proof,
2. the contract emits `ExploitFound` for the correct beneficiary,
3. an internal transfer of 1000 wei to that address occurs in the same block, confirming that proof verification and bounty payment are settled atomically in a single transaction.

This live transaction proves end-to-end that an auditor, armed only with the bytecode of a vulnerable contract, the JSON snapshot, and the `program_specification`, can convince an on-chain verifier of exploitability *without disclosing a single opcode of the attack path.* The whole proof, including the bounty payment, fits in one Ethereum transaction and costs 304,570 gas on Holesky.

Chapter 4

Performance Evaluation and Results

This chapter explains *how* we measure the performance of *zkpoex* and *what* the measurements reveal. Only the execution that runs *inside* the RISC Zero zkVM is timed; the subsequent on-chain Groth16 verification is ignored because its cost depends on gas price and network congestion. Every run is profiled with the official RISC Zero workflow [76], and the resulting data are rendered as Brendan Gregg-style flame graphs for fast, visual hotspot discovery [77].

4.1 Evaluation Metrics

This section defines the very small set of numbers and pictures we rely on when comparing different proofs. In the next subsections, we will see all the evaluation metrics calculated following the *Guest Profiling Guide* from the Risc Zero Docs [76].

4.1.1 Quantitative metric

As a main quantitative metric, there are the *total cycles*: the sum of guest and kernel cycles, which serves as the single numeric predictor of prover cost, because the RISC Zero prover scales linearly with this length [78]. Wall-clock seconds are logged for context, but fluctuate with host frequency and cache hierarchy, so they are never used for cross-machine comparisons.

4.1.2 Flame graph metric

A flame graph is an SVG where each rectangle represents a stack frame sampled at every cycle; the *width* of the box equals the fraction of total cycles

spent in that frame, and the x-axis therefore aggregates 100% of runtime [77]. The y-axis shows call-stack depth, making deep towers easy to spot as recursive or interpreter loops [77]. Colours are random warm tones with no semantics. With the flame graph, all the data is on screen at once, and the hottest code-paths are immediately obvious as the widest functions. The built-in `pprof` viewer adds hover tool-tips, zoom, regex search, and red/blue differential mode for regressions [79, 80].

4.2 Profiling Methodology

This section records the exact steps needed to reproduce the profiles shown later. We calculated a *profile* for each case study in the Case Study Section in Chapter 3. Proof generation times will be shown in the next section 4.3.

4.2.1 Prerequisites

As indicated in [76], the main prerequisite for viewing flame graphs and all related metrics is Go. So you need to install it because it bundles with it the *pprof* tool. Obviously, all the prerequisites of the Risc Zero zkVM are necessary.

After installing Go, you need to set all these variables as below:

```
RISC0_PPROF_OUT=./profile.pb \
RISC0_DEV_MODE=1 \
RUST_LOG=info
```

The variable `RISC0_DEV_MODE=1` skips real proving so the profile is generated in seconds [81]. When this flag is enabled the zkVM runs the guest exactly as in production but returns a *fake receipt* instead of executing the heavyweight STARK and Groth16 pipelines.

Generating the cryptographic proof is unnecessary at this stage because we are interested solely in the guest execution time and the cycle distribution recorded inside the zkVM trace; both are fully captured before the heavy STARK computation and Groth16 compression begin. Dev-mode, therefore, produces an identical flame graph and cycle count in a fraction of the time, giving an accurate snapshot of run-time performance without incurring the extra cost of creating a final, on-chain-ready proof.

After setting the variables, simply run the `just` command to generate an exploit proof for any example you want to demonstrate (see Listing 3.3 as an example, in this case for the *BasicVulnerable* case study), and the `profile.pb` file will be generated and saved in the location indicated in the `RISC0_PPROF_OUT` variable.

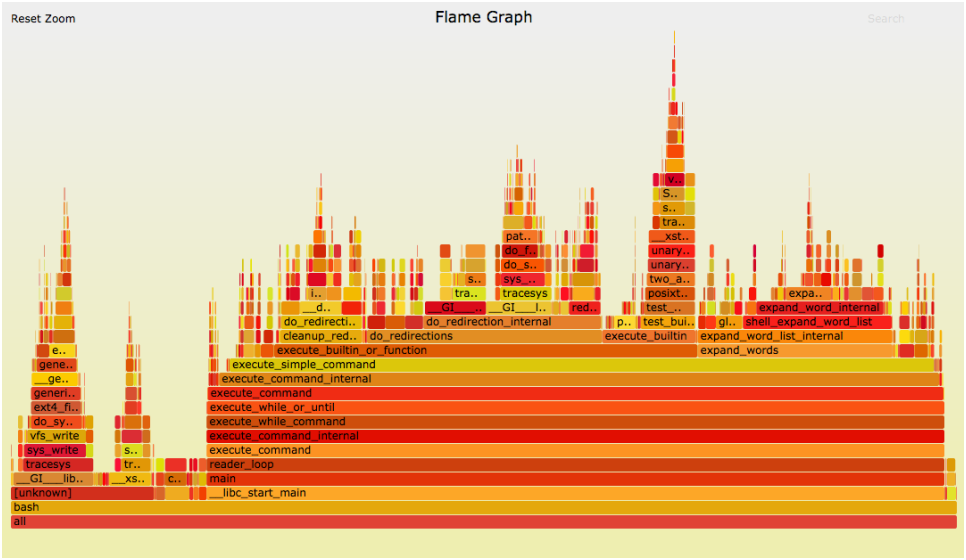

Figure 4.1: Interactive flame graph generated by `pprof`.

4.2.2 Visualisation

Once the profiling run has produced its `profile.pb`, the file can be inspected directly with the web-UI bundled into `pprof`.

Launching

```
go tool pprof -http=127.0.0.1:8000 ./profile.pb
```

starts a local server on port 8000; visiting the indicated URL opens an interactive SVG that embeds every cycle of execution. The interface is intentionally minimal yet powerful: moving the cursor across the graph reveals, in a tooltip, the fully qualified function name together with the exact number of cycles it consumed, so the dominant hot spots become obvious without leaving the page. Clicking any rectangle zooms the view so that the selected frame now fills the width of the browser window, effectively filtering the graph to the subtree beneath that call; a breadcrumb trail at the top of the page lets you ascend the stack again with a single click.

If you have two profiles, for example, before and after an optimisation, you can feed both *protobufs* to the same `pprof` command; the viewer then switches automatically to differential mode and colours frames according to how much their share of total cycles has grown or shrunk, a visual aid that makes regressions stand out immediately.

Because the flame graph is a self-contained SVG, it can be downloaded with the browser's "Save" dialog; no external CSS or JavaScript is required, so anyone can view the graph without installing additional tools [79, 80].

In Figure 4.1, an example of a flame graph.

4.3 Profiling Analysis

This section applies the profiling workflow to the three exploits introduced in Case Study Section in Chapter 3, concentrating on the `run_evm` frame because all vulnerability logic executes inside the EVM interpreter embedded in the zkVM. For each case study, we report the total guest cycle budget, discuss how many of those cycles fall under `run_evm`, and highlight the most expensive descendants.

The table below converts the raw "guest-cycle" counters emitted by `pprof` into an *ideal* execution time via

$$\text{time}_{\text{guest}} = \frac{\text{cycle count}}{f_{\text{CPU}}}, \qquad f_{\text{CPU}} = 2.0 \times 10^9 \text{ Hz}$$

(f_{CPU} is the base clock of a Ryzen 7 4700U). Because one guest tick is mapped to a single `rv32im` instruction, the division gives a *lower-bound* latency: stalls, cache misses, and the subsequent STARK / Groth16 stages are *not* included here. For instance, the *Basic Vulnerable* run consumes 702 098 ticks, which yields $\frac{702\,098}{2 \cdot 10^9} \approx 0.35$ ms.

Case study	**total cycles**	`run_evm` **share**	**guest time (ms)***
Basic Vulnerable	9.99×10^5	70.3 %	0.35
Unchecked Arithmetic	4.05×10^5	75.9 %	0.15
Reentrancy Drain	1.81×10^6	91.2 %	0.82

Table 4.1: Cycle budget, dominance of `run_evm`, and ideal guest latency on an AMD Ryzen 7 4700U.

* *Guest time* covers only the RISC-V trace that executes the EVM logic; it excludes circuit interpolation, FRI commitment, Groth16 compression, and I/O.

4.3.1 BasicVulnerable Case Study Evaluation

Figure 4.2 shows that `run_evm` consumes approximately 70 % of all cycles. Inside that frame, most time is spent hashing the pre-state (`context_state`) and calculating program-spec digests, followed by the Keccak `p1600` primitive. The actual EVM execution path (`transact_call` →`Runtime::run`) is comparatively shallow because the exploit issues a single value-bearing `CALL` that forwards the contract's entire balance to `address(0)`; once the transfer is executed, no further storage writes or complex gas-cost paths are triggered, so the interpreter exits quickly.

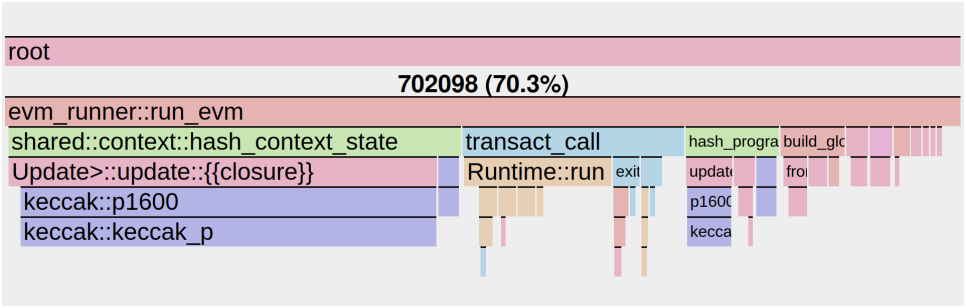

Figure 4.2: Flame graph for the *Basic Vulnerable* logical exploit.

4.3.2 Unchecked Arithmetic Case Study Evaluation

Figure 4.3 confirms that `run_evm` still dominates the trace, yet its internal profile differs markedly from the Basic case. The interpreter loop is wider even though the total cycle budget is smaller, because the exploit executes two storage touching opcodes: one `SLOAD` to fetch `balance` and one `SSTORE` to write back the wrapped value. Each storage access triggers the Merkle-Patricia-Trie machinery inside `StackExecutor::transact_call`, which explains the pronounced block of hashing just beneath the loop. The unchecked subtraction itself appears as a narrow frame; the real cost lies in updating the trie and recalculating the slot digest. A `run_evm` share of 75.9%, therefore, means JSON decoding and context hashing are amortised over fewer total cycles, making the interpreter's storage paths the clear target for future optimisation.

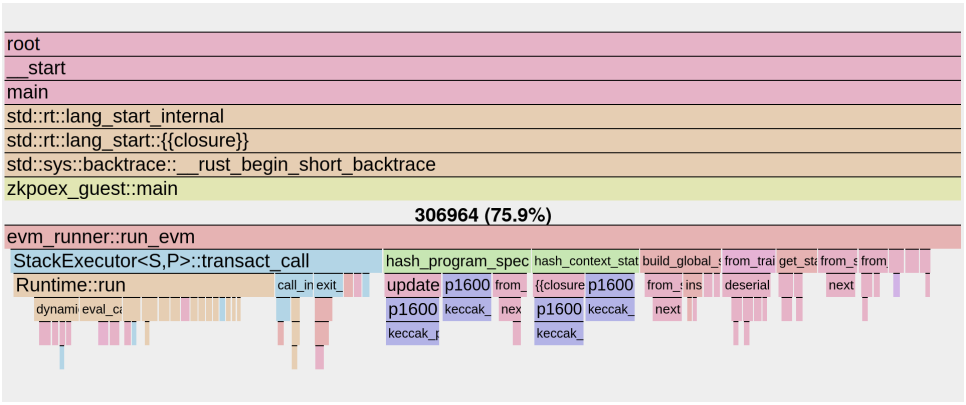

Figure 4.3: Flame graph for the *Unchecked Arithmetic* overflow exploit.

4.3.3 Reentrancy Drain Evaluation

Figure 4.4 shows that `run_evm` dominates 91.2 % of the total trace, far higher than in the other two runs. The reason is visible in the shape of the graph: every external `CALL` to the attacker's fallback re-enters `withdrawETH`, so the interpreter never unwinds to the host; instead, it spawns a nested EVM invocation that appears as another wide block under the parent `eval` frame. Because the balance mapping is cleared only after the transfer, each callback observes a positive balance and triggers yet another `CALL`, producing a forest of stacked frames. The dynamic gas-cost routine (`dynamic_opcode_cost`) forms a distinct tower to the right, reflecting the variable cost charged for cold versus warm storage slots on every entry. Even with this heavier control flow, the timing scales linearly: the trace contains roughly twice the cycles of *Basic Burn* and takes roughly twice as long to execute, confirming that the total cycle metric reliably forecasts wall clock performance in reentrancy scenarios.

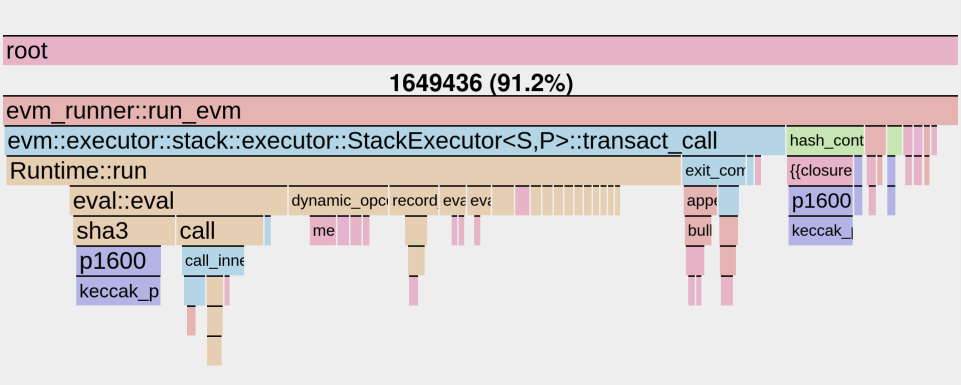

Figure 4.4: Flame graph for the *Reentrancy Drain* exploit.

Chapter 5

Conclusions and Future Perspectives

The idea at the heart of *zkpoex* was sparked by a long-standing pain point in the Ethereum security scene: responsible disclosure is hard. Researchers worry about exposing too much of an exploit, project teams often respond slowly, or not at all, and bug-bounty agreements can collapse into distrust. In DeFi, where immutable contracts control vast sums of money, that friction becomes dangerous: an unpatched bug can wipe out liquidity in minutes. Our goal was, therefore, to build a neutral handshake between attacker and defender, one that proves a vulnerability exists without revealing how it works.

The prototype shows that zero-knowledge receipts can certify the *existence* of an exploit on real mainnet bytecode while hiding all private inputs. The researcher produces a journal plus a 1kb Groth16 seal; the project owner verifies the receipt on-chain and sees only a Boolean flag and two Keccak hashes. No calldata, no control-flow trace, no storage diff, yet the proof remains infallible.

Only one earlier prototype, zkPoEX [20], an ETH Denver hackathon PoC, attempted a similar idea as said in this thesis, but it was not yet generalizable as a tool to prove arbitrary exploits and offered no general condition engine or cycle-level profiling, making *zkpoex* the first comprehensive step toward truly "provable exploits".

What makes the system technically novel? Rather than proving a whole block or a generic computation, *zkpoex* pins a single EVM trace into the RISC Zero zkVM and evaluates user-supplied safety conditions inside the circuit.

Two directions look particularly promising. First, a formal model of the conditions engine could be submitted to an academic venue, providing a machine-checked guarantee that "condition true → exploit real" holds un-

der all circumstances. Second, linking the receipt format with a bug-bounty escrow contract would let rewards flow automatically once a proof is verified, removing manual negotiations. Additionally, by transforming exploit disclosure into a private yet verifiable transaction, *zkpoex* realigns incentives: researchers retain ownership of their discoveries, projects receive irrefutable evidence, and the bug bounty platforms can also solve spam issues. The prototype thus hints at a future where offensive research and defensive engineering cooperate under the umbrella of zero-knowledge cryptography, making DeFi both safer and more collaborative.

References

[1] Ethereum Foundation. Accounts, transactions, gas and fees. https://ethereum.org/en/developers/docs/accounts/, 2023.

[2] Takenobu T. Ethereum evm illustrated. https://takenobu-hs.github.io/downloads/ethereum_evm_illustrated.pdf, 2018.

[3] Zerocap. London hard fork: The long-term ramifications. https://zerocap.com/insights/research-lab/london-hard-fork-long-term/, 2023.

[4] Zk camp. https://www.zkcamp.xyz.

[5] Vitalik Buterin. How do trusted setups work? https://vitalik.eth.limo/general/2022/03/14/trustedsetup.html.

[6] Z.K. Shen. Intro to zkvm? https://www.lita.foundation/blog/zero-knowledge-paradigm-zkvm, 2024.

[7] zkpoex docs. https://github.com/ziemen4/zkpoex/blob/main/docs/0.1.0/DOCS.md, 2025.

[8] Risc0 Docs. zkvm overview. https://dev.risczero.com/api/zkvm/, 2025.

[9] Risc0 Docs. Proof system overview. https://github.com/risc0/risc0/blob/main/website/docs/proof-system/proof-system.md, 2025.

[10] Chainalysis. *$2.2 billion stolen from crypto platforms in 2024, but hacked volumes stagnate toward year-end as dprk slows activity post-july*. https://www.chainalysis.com/blog/crypto-hacking-stolen-funds-2025/.

[11] Alessandro Cavaliere (galexela) and ziemann. *zkpoex*. https://github.com/ziemen4/zkpoex.

[12] Matthew Green, Mathias Hall-Andersen, Eric Hennenfent, Gabriel Kaptchuk, Benjamin Perez, and Gijs Van Laer. Efficient proofs of software exploitability for real-world processors. *Proceedings on Privacy Enhancing Technologies*, 2023.

[13] Santiago Cuéllar, Bill Harris, James Parker, Stuart Pernsteiner, and Eran Tromer. Cheesecloth: Zero-knowledge proofs of real-world vulnerabilities. https://arxiv.org/abs/2301.01321, 2023.

[14] Antti Valmari. The state explosion problem. In Wolfgang Reisig and Grzegorz Rozenberg, editors, *Lectures on Petri Nets I: Basic Models: Advances in Petri Nets*, pages 429–528, Berlin, Heidelberg, 1998. Springer Berlin Heidelberg.

[15] Risc zero: Universal zero knowledge. https://risczero.com/.

[16] Pure rust implementation of ethereum virtual machine. https://github.com/rust-ethereum/evm.

[17] Barbara Kitchenham, Pearl Brereton, David Budgen, Mark Turner, John Bailey, and Stephen Linkman. Systematic literature reviews in software engineering-a systematic literature review. *Information and Software Technology*, 51:7–15, 01 2009.

[18] Swanky mac-n-cheese library. https://github.com/GaloisInc/swanky/tree/dev/mac-n-cheese.

[19] Jens Ernstberger, Stefanos Chaliasos, Liyi Zhou, Philipp Jovanovic, and Arthur Gervais. Do you need a zero knowledge proof? *Cryptology ePrint Archive*, 2024.

[20] zkoranges and federava. *ZkPoEX*. https://github.com/zkoranges/zkPoEX/.

[21] Inas Hasnaoui, Maria Zrikem, and Rajaa Elassali. Beyond the bug bounty programs trilemma: Bounty 3.0's blockchain-zkp approach. In *2023 7th IEEE Congress on Information Science and Technology (CiSt)*, pages 663–668, 2023.

[22] D. Chaum. *Computer Systems Established, Maintained, and Trusted by Mutually Suspicious Groups*. PhD thesis, University of California, Berkeley, 1982. https://evervault.com/papers/chaum.pdf.

[23] Stuart Haber and W. Scott Stornetta. How to time-stamp a digital document. *J. Cryptology*, 3:99–111, 1991.

[24] Dave Bayer, Stuart Haber, and W. Scott Stornetta. Improving the efficiency and reliability of digital time-stamping. In Renato Capocelli, Alfredo De Santis, and Ugo Vaccaro, editors, *Sequences II*, pages 329–334, New York, NY, 1993. Springer New York.

[25] Wei Dai. b-money. https://www.weidai.com/bmoney.txt, 1998. Accessed: 2025-06-27.

[26] H. Finney. Reusable proof of work. https://cryptome.org/rpow.htm.

[27] S. Nakamoto. Bitcoin: A peer-to-peer electronic cash system. https://bitcoin.org/bitcoin.pdf, 2008.

[28] W. Diffie and M. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, 22(6):644–654, 1976.

[29] Cesar Castellon, Swapnoneel Roy, Patrick Kreidl, Ayan Dutta, and Ladislau Bölöni. Energy efficient merkle trees for blockchains. In *2021 IEEE 20th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*, pages 1093–1099, 2021.

[30] Ethereum Foundation. Proof-of-stake (pos) validators. https://ethereum.org/en/developers/docs/consensus-mechanisms/pos/#validators, 2024.

[31] Vitalik Buterin, Diego Hernandez, Thor Kamphefner, Khiem Pham, Zhi Qiao, Danny Ryan, Juhyeok Sin, Ying Wang, and Yan X Zhang. Combining ghost and casper, 2020.

[32] Bitcoin developer guide. https://developer.bitcoin.org/devguide/block_chain.html.

[33] Joseph Poon and Thaddeus Dryja. The bitcoin lightning network: Scalable off-chain instant payments. https://lightning.network/lightning-network-paper.pdf, 2016. White paper.

[34] Malte Möser, Rainer Böhme, and Dominic Breuker. An inquiry into money laundering tools in the bitcoin ecosystem. In *2013 APWG eCrime Researchers Summit*, pages 1–14. IEEE, 2013.

[35] Vitalik Buterin et al. A next-generation smart contract and decentralized application platform. https://ethereum.org/en/whitepaper/.

[36] Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger. https://ethereum.github.io/yellowpaper/paper.pdf, 2014. Ethereum Yellow Paper.

[37] Ethereum Foundation. Eip-2718: Typed transaction envelope. https://eips.ethereum.org/EIPS/eip-2718, 2020.

[38] Ethereum Foundation. Eip-2930: Optional access lists. https://eips.ethereum.org/EIPS/eip-2930, 2020.

[39] Ethereum Foundation. Eip-1559: Fee market change for eth 1.0 chain. https://eips.ethereum.org/EIPS/eip-1559, 2021.

[40] S Goldwasser, S Micali, and C Rackoff. The knowledge complexity of interactive proof-systems. In *Proceedings of the Seventeenth Annual ACM Symposium on Theory of Computing*, STOC '85, page 291–304, New York, NY, USA, 1985. Association for Computing Machinery.

[41] Amos Fiat and Adi Shamir. How to prove yourself: Practical solutions to identification and signature problems. In Andrew M. Odlyzko, editor, *Advances in Cryptology — CRYPTO' 86*, pages 186–194, Berlin, Heidelberg, 1987. Springer Berlin Heidelberg.

[42] Manuel Blum, Paul Feldman, and Silvio Micali. Non-interactive zero-knowledge and its applications. In *Proceedings of the Twentieth Annual ACM Symposium on Theory of Computing*, STOC '88, page 103–112, New York, NY, USA, 1988. Association for Computing Machinery.

[43] Nir Bitansky, Ran Canetti, Alessandro Chiesa, and Eran Tromer. From extractable collision resistance to succinct non-interactive arguments of knowledge, and back again. In *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference*, ITCS '12, page 326–349, New York, NY, USA, 2012. Association for Computing Machinery.

[44] Bryan Parno, Jon Howell, Craig Gentry, and Mariana Raykova. Pinocchio: Nearly practical verifiable computation. In *2013 IEEE Symposium on Security and Privacy*, pages 238–252, 2013.

[45] Jens Groth. On the size of pairing-based non-interactive arguments. In Marc Fischlin and Jean-Sébastien Coron, editors, *Advances in Cryptology – EUROCRYPT 2016*, pages 305–326, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg.

[46] Benedikt Bünz, Jonathan Bootle, Dan Boneh, Andrew Poelstra, Pieter Wuille, and Greg Maxwell. Bulletproofs: Short proofs for confidential transactions and more. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 315–334, 2018.

[47] Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. Scalable, transparent, and post-quantum secure computational integrity. Cryptology ePrint Archive, Report 2018/046, 2018.

[48] Ariel Gabizon, Zachary J. Williamson, and Oana Ciobotaru. Plonk: Permutations over lagrange-bases for oecumenical noninteractive arguments of knowledge. Cryptology ePrint Archive, Report 2019/953, 2019.

[49] Sean Bowe, Jack Grigg, and Daira Hopwood. Halo: Recursive proof composition without a trusted setup. Cryptology ePrint Archive, Report 2019/1021, 2019.

[50] Srinath Setty. Nova: Recursive zero-knowledge arguments from folding schemes. Cryptology ePrint Archive, Report 2021/370, 2021.

[51] Benedikt Bünz and Binyi Chen. Protostar: Generic efficient accumulation/folding for special-sound protocols. Cryptology ePrint Archive, Report 2023/620, 2023.

[52] Polygon Labs. Introducing plonky2. https://polygon.technology/blog/introducing-plonky2, 2022.

[53] Nojan Sheybani, Anees Ahmed, Michel Kinsy, and Farinaz Koushanfar. Zero-knowledge proof frameworks: A survey. arXiv preprint arXiv:2502.07063, 2025.

[54] Ulrich Haböck, David Levit, and Shahar Papini. Circle STARKs. Cryptology ePrint Archive, Paper 2024/278, 2024.

[55] Jean-Jacques Quisquater, Myriam Quisquater, Muriel Quisquater, Michaël Quisquater, Louis Guillou, Marie Annick Guillou, Gaïd Guillou, Anna Guillou, Gwenolé Guillou, and Soazig Guillou. How to explain zero-knowledge protocols to your children. In Gilles Brassard, editor, *Advances in Cryptology — CRYPTO' 89 Proceedings*, pages 628–631, New York, NY, 1990. Springer New York.

[56] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and Understanding Bugs in C/C++ Compilers. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 283–294, 2011.

[57] Electric Coin Company. Explaining halo 2. https://electriccoin.co/blog/explaining-halo-2/, 2020.

[58] Eli Ben-Sasson, Iddo Bentov, Yossi Horesh, and Michael Riabzev. Interactive oracle proofs and polynomial commitments, 2022.

[59] Lior Goldberg, Shahar Papini, and Michael Riabzev. Cairo – a turing-complete STARK-friendly CPU architecture. Cryptology ePrint Archive, Paper 2021/1063, 2021.

[60] Nethermind. Announcing cairo 1.0: safer syntax, richer tooling. https://medium.com/nethermind-eth/a-first-look-at-cairo-1-0-a-safer-stronger-simpler-provable-programming-language-892ce4c07b38, 2022. Blog post.

[61] StarkWare Industries. Starknet alpha on mainnet: a permissionless, decentralized zk-rollup. https://medium.com/starkware/starknet-alpha-now-on-mainnet-4cf35efd1669, 2021. Blog post.

[62] RISC Zero. Designing high-performance zkvms. https://risczero.com/blog/designing-high-performance-zkVMs, 2024.

[63] txFusion Team. zksync era mainnet alpha - what's that? https://medium.com/tx-fusion/zksync-era-mainnet-alpha-whats-that-951c83e8a3d7, 2023. Blog post.

[64] Succinct Labs. Introducing sp1: A performant, 100 https://blog.succinct.xyz/introducing-sp1/, 2024. Technical blog.

[65] Polygon Labs. Miden is the edge blockchain. https://miden.xyz/, 2024.

[66] Risc0 Docs. Verifier contracts. https://dev.risczero.com/api/blockchain-integration/contracts/verifier, 2025.

[67] Risc0 Docs. Receipt 101. https://dev.risczero.com/api/zkvm/receipts, 2025.

[68] Risc0 Docs. Key terminology. https://dev.risczero.com/terminology, 2025.

[69] Risc0 Docs. Remote proving using bonsai. https://dev.risczero.com/api/generating-proofs/remote-proving, 2025.

[70] Risc0 Docs. Host code 101. https://dev.risczero.com/api/zkvm/host-code-101, 2025.

[71] RISC Zero. Designing high-performance zkvms. https://risczero.com/blog/designing-high-performance-zkVMs, 2023. Section *"Name Dropping: The 0STARK Protocol"*.

[72] Rust-Ethereum Project. Pure rust implementation of ethereum virtual machine. https://github.com/rust-ethereum/evm, 2024.

[73] Parity Technologies. `MemoryAccount` in `evm::backend` from evm crate docs. https://docs.rs/evm/0.42.0/evm/backend/struct.MemoryAccount.html, 2024.

[74] Parity Technologies. `StackExecutor` from evm crate docs. https://docs.rs/evm/latest/evm/executor/stack/index.html, 2024.

[75] Solidity Team. Solidity 0.8.x preview. https://soliditylang.org/blog/2020/10/28/solidity-0.8.x-preview, 2020.

[76] RISC Zero. Guest profiling guide. https://dev.risczero.com/api/zkvm/profiling.

[77] Brendan Gregg. Cpu flame graphs. https://www.brendangregg.com/FlameGraphs/cpuflamegraphs.html.

[78] RISC Zero. Guest optimization guide. https://dev.risczero.com/api/zkvm/optimization.

[79] Google. pprof web ui — flame graph. https://github.com/google/pprof/issues/166.

[80] Brendan Gregg. Differential flame graphs. https://www.brendangregg.com/blog/2014-11-09/differential-flame-graphs.html.

[81] RISC Zero. Dev-mode receipts. https://dev.risczero.com/api/generating-proofs/dev-mode.